(2)

Revised Final Report:

CMS-2 Reverse Engineering & ENCORE/SRE Integration

Contract # N00014-91-C-0240

May 1992
Revised July 1992

General Electric Company
Corporate Research and Development
P. O. Box 8
Schenectady, NY 12301

This work was supported in part by
Naval Surface Warfare Center (NSWCDD)
under contract #N00014-91-C-0240
with the Office of Naval Research

Table of Contents

CMS-2 Reverse Engineering and ENCORE/SRE Integration Study Revised Final Report

Part I CMS-2 Reverse Engineering Technology 1

Chapter 1.0 Technology Overview 1

Chapter 2.0 Automatic Operation, Installation, and Setup 6

2.1 Automatic Operation 6

2.2 RET Installation 8

2.3 System Setup 8

Chapter 3.0 Information Extraction 9

3.1 User Perspective 9

3.2 Internals 9

Chapter 4.0 Comment Processing 14

4.1 User Perspective 14

4.2 Internals 14

Chapter 5.0 System Integration 15

5.1 User Perspective 15

5.2 Internals 15

Chapter 6.0 Building the Teamwork/SD Reverse Engineering Database 16

6.1 User Perspective 16

6.2 Internals 16

Chapter 7.0 TeamWork Environment 16

7.1 Invocation 16

7.2 Basic Teamwork Displays 16

7.3 GE-Supplied Extensions 17

Chapter 8.0 File Formats 18

8.1 Middle Files 18

8.2 Comment Files 20

Part II ENCORE-SRE Integration Study 21

Appendix A: CMS RET User's Manual

Appendix B: Introduction to ENCORE Internal Representation

Appendix C: ADL Description of the Ada Internal Representation

Appendix D: Introduction to the ENCORE Symbol Table

Appendix E: SRE/ESL Internal Representation

CMS-2 Reverse Engineering and ENCORE/SRE Integration Study

Revised Final Report

This is the final report for the contract N00014-91-C-0240. It is divided into two parts: one addressing the CMS-2 Reverse Engineering Technology, and the other the ENCORE/SRE (Software Reengineering Environment[1]) Integration Study.

Part I CMS-2 Reverse Engineering Technology

Part I presents an overview of the CMS-2 Reverse Engineering Technology (CMS RET) produced for this contract. It includes a description of the operation of the tool, as well as the work done, and the portions reused from other projects. Chapter 1.0 gives an overview of the work done, Chapter 2.0 presents installation information and recommended operation instructions, Chapters 3.0 through 7.0 provide detailed discussions of the functional areas involved, and Chapter 8.0 details the formats of two files which are crucial to anyone customizing or extending CMS RET. Appendix A contains the User Manual for CMS RET.

Chapter 1.0 Technology Overview

The work done for this contract demonstrates that:

- Automated extraction of design information from an existing software system written in CMS-2 can be used to document that system *as-built*, and that
- The extracted information can be entered into the database of a commercially available CASE tool and manipulated via the CASE interface.

The delivered prototype operates on Sun/4 workstations and interfaces to the Cadre Team*work*/SD[2] and Cadre Team*work*/C Rev CASE tools. (If an interface to another CASE tool should be required, the Database Generator would be reimplemented. Chapter 8.0 provides the format of this phase's input files, so such an effort would be fairly well defined.) The delivered prototype handles a subset of the dialect of CMS-2 known as CMS-2L. This subset is detailed in *Detailed Design for CMS-2 to Ada Translation.*[3]

---

1. SRE (Software Reengineering Environment) is a set of reengineering tools being constructed by Computer Command and Control Company, Philadelphia, PA.
2. Team*work*/SD and Teamwork/C Rev are a registered trademarks of Cadre Technologies Inc.
3. *Detailed Design for CMS-2 to Ada Translation*, January 1992, GE internal document.

The key features of the CMS RET system are:

- The interactive visual interface to the extracted information is provided by a commercially available CASE tool.
- Information describing software system design is automatically extracted from source files and organized in a language independent standard mode.
- A method has been developed which exploits project-specific commenting conventions in order to automatically extract comments to the database.

There are three major vehicles of communication provided by the Team*work*/SD interface:

- Structure charts illustrate the calling relationships between modules. (see Figure 1)
- Module specifications (Mspecs) present a description of each module. (see Figure 2)
- Data dictionary entries (DDEs) describe the global variables. (see Figure 3)

In addition the system will answer the following questions:

- Where is this variable referenced?
- Which modules call this one?

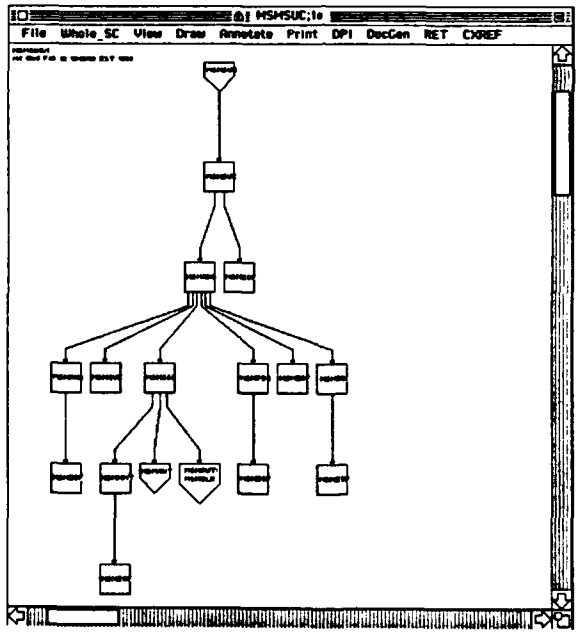


FIGURE 1. Structure Chart generated by CMS RET, displayed by Team*work*/SD

The Structure Chart displays the calling relationships between the modules of the system, providing a convenient method of traversing the software under examination. It provides direct access to the Mspecs of the modules displayed, as well a facility for looking up the call tree of any module displayed.

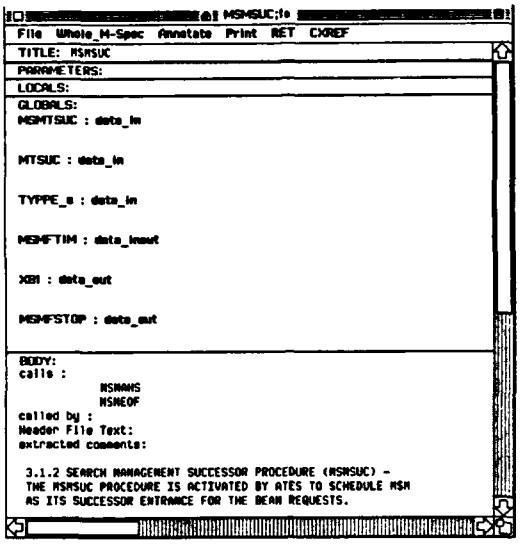


FIGURE 2. Mspec generated by CMS RET, displayed by Team*work*/SD

The Mspec describes the important aspects of a module: its parameters, the names of any global variables which it references, and comments extracted from its source code. It also summarizes the structure chart information which is pertinent to the module being displayed, and provides direct access to the relevant source code. From the Mspec for a given module, the user can easily move to the Mspecs of related modules and the DDE's for any relevant global variables.

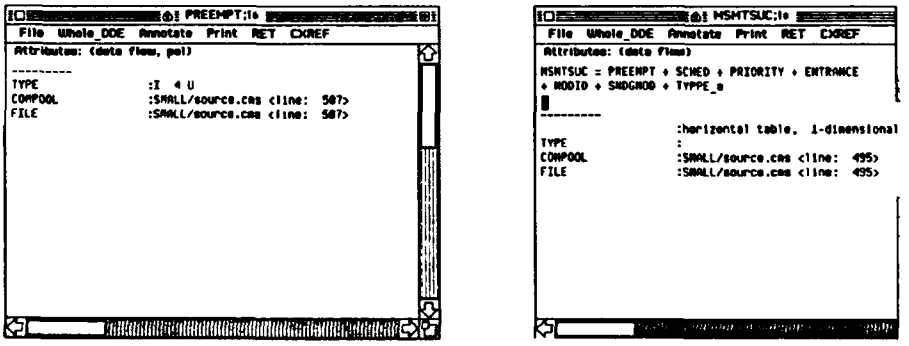


FIGURE 3. Two examples of DDEs produced by CMS RET, displayed by Team*work*/SD

The DDE describes a global data item, displaying its structure and type, as well as the location of its declaration (displayed with and without reference to any *include* files). If the item in question is a table, its DDE gives the user immediate access to its fields' DDEs. In addition, the DDE provides direct access to the relevant source code, and permits the user to display a list of modules which access the data item in question.

The objective of the CMS-2 Reverse Engineering Technology is to provide the information for the displays in Figures 1, 2, and 3. To achieve this, GE built upon two past projects: a

CMS-2 to Ada translator (CMS2Ada) [1] and a Jovial Reverse Engineering Technology (JRET) [2]. CMS2Ada provided CMS-2 language capabilities which could be reused, and JRET provided a framework for a general reverse engineering technology which could be adapted to fit CMS-2.

The CMS-2 Reverse Engineering Technology is made up of four functional areas: 1) Information Extraction, 2) Comment Processing, 3) Database Generation, and the 4) Cadre Team*work* interface. The first two functions (Information Extraction and Comment Processing) operate on a file-by-file basis, collecting relevant information into a language-independent format. Database Generation builds a system-wide view of the information, writing it into a form which Team*work* can process. The final functional area is Cadre Team*work*/SD. These four functions work together to visually present as-built architectural information about an existing system. Figure 4 shows how these areas fit together.

The bulk of the work for this contract was done on the first two functional areas: Information Extraction and Comment Processing. The Database Generator was reused from JRET and Team*work*/*SD* is a commercial product from Cadre, which was extended using their extensible interface[1]. (These extensions were also reused from JRET.)

Statement A per telecon James Smith
ONR /Code 1267
Arlington, VA 22217-5000

NWW 7/15/92

1. See *Teamwork User Menus User's Guide*, Chapters 2 and 3 for a detailed description of this facility.

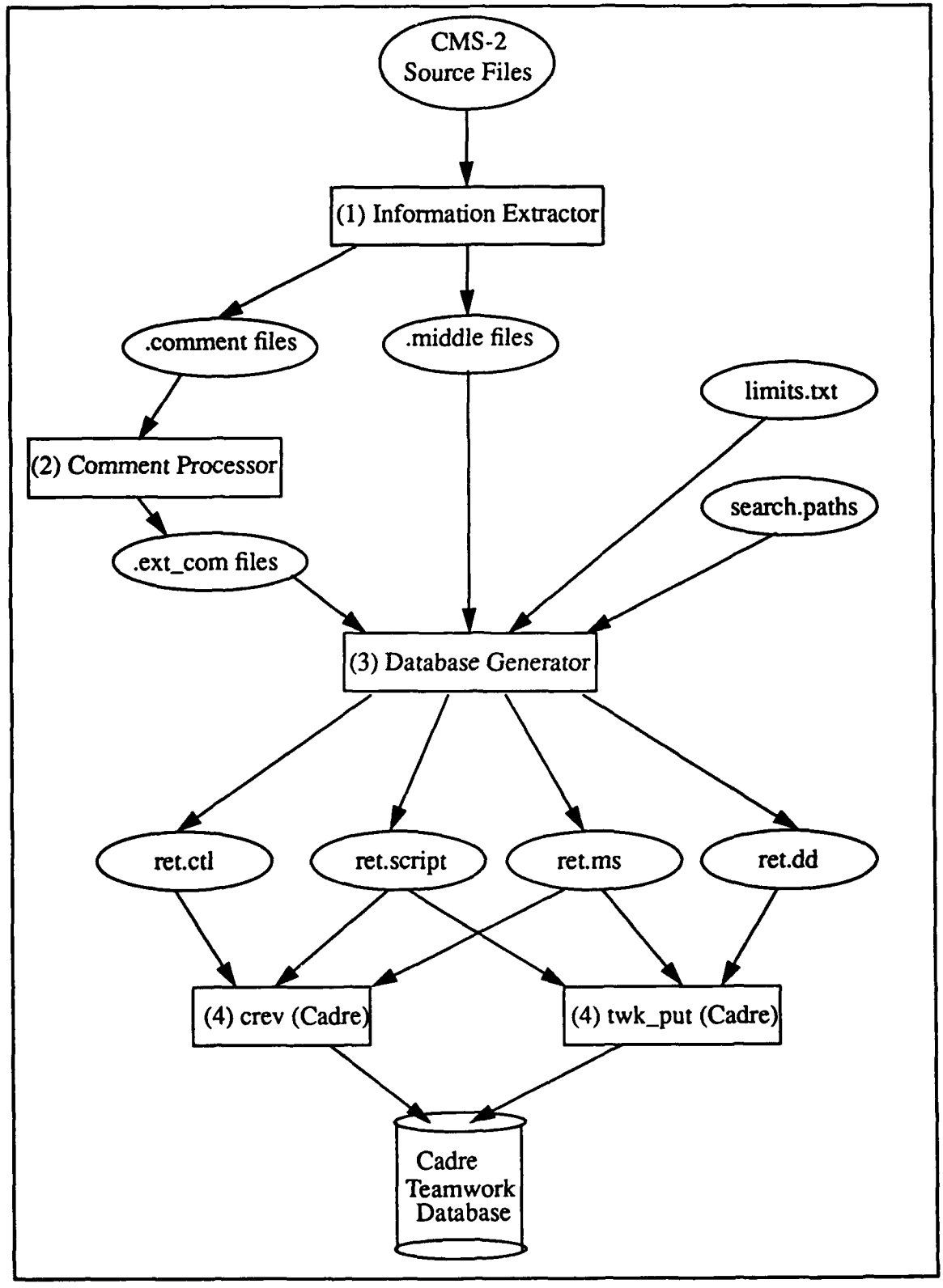


FIGURE 4. System Overview of CMS RET

Chapter 2.0 Automatic Operation, Installation, and Setup

2.1 Automatic Operation

CMS RET is run as a series of steps. These steps are usually run across all Computer Software Configuration Items[1] (CSCIs) in a CMS-2 system when the initial build of the Team*work* database is done. (See Section 2.3 for further details about CSCIs.) Over time, as files change, there may be a need to rebuild the database. If only a small number of CSCIs have been affected, it may be preferable to run rebuild only on the part of the database dealing with the affected CSCI's.

The $RET_DB_HOME/admin/build-ret script will run all the steps, either for one CSCI or for the whole CMS-2 system. It takes care of all the details and housekeeping involved, and produces log files so that the user can monitor its progress. It is invoked as follows:

$RET_DB_HOME/admin/build-ret [*n* [*CSCI_name*]]

n - is an integer between 1 and 7 specifying which operation is to be performed. If it is not entered on the command-line, build-ret prompts for an input. The choices are as follows:

1	CMS Rev pass 3	(information extraction)
2	CMS Rev Comment Processing	
3	CMS Rev pass 4	(system integration, part 1)
4	Post-Process CMS Rev	(system integration, part 2)
5	Create TeamWork Database	
6	Dump TeamWork Database	
7	Restore TeamWork Database	

CSCI_***name***- indicates the CSCI on which the specified operation should be performed. If omitted, the processing will affect all CSCI's, as determined by the contents of $RET_DB_-HOME/src/search.paths.

In normal operation, one would call the script with option 1, then 2, and so on, until option 5 had been performed. A CSCI name is not generally specified unless a particular CSCI is being rebuilt separately for some reason.

Here is the sequence of commands and system responses which would be issued to build a full CMS-2 system which contains the CSCIs COLLECT and ANALYZE:

```
> $RET_DB_HOME/admin/build-ret 1
Begin RET Build Program, Tue Apr 21 10:01:36 EDT 1992
ANALYZE         CMS Rev Pass 3 Tue Apr 21 10:01:39 EDT 1992
COLLECT         CMS Rev Pass3 Tue Apr 21 10:01:54 EDT 1992
End RET Build Program, Tue Apr 21 10:02:14 EDT 1992
> $RET_DB_HOME/admin/build-ret 2
Begin RET Build Program, Tue Apr 21 10:02:28 EDT 1992
```

1. See DOD-STD-2167A, June 4, 1985, for definitions relevant to Computer Software Organization.

```
ANALYZE    CMS Rev Comment Processing Tue Apr 21 10:02:31 EDT 1992
COLLECT    CMS Rev Comment Processing Tue Apr 21 10:02:42 EDT 1992
End RET Build Program, Tue Apr 21 10:02:50 EDT 1992
> $RET_DB_HOME/admin/build-ret 3
Begin RET Build Program, Tue Apr 21 10:03:03 EDT 1992
ANALYZE    CMS Rev Pass 4a Tue Apr 21 10:03:05 EDT 1992
COLLECT    CMS Rev Pass 4a Tue Apr 21 10:03:08 EDT 1992
      CMS Rev Pass 4b Tue Apr 21 10:03:12 EDT 1992
ANALYZE    CMS Rev Pass 4c Tue Apr 21 10:03:15 EDT 1992
COLLECT    CMS Rev Pass 4c Tue Apr 21 10:03:23 EDT 1992
End RET Build Program, Tue Apr 21 10:03:33 EDT 1992
> $RET_DB_HOME/admin/build-ret 4
Begin RET Build Program, Tue Apr 21 10:03:49 EDT 1992
ANALYZE    CMS Rev Post-Processing Tue Apr 21 10:03:51 EDT 1992
COLLECT    CMS Rev Post-Processing Tue Apr 21 10:03:55 EDT 1992
End RET Build Program, Tue Apr 21 10:03:59 EDT 1992
> $RET_DB_HOME/admin/build-ret 5
Begin RET Build Program, Tue Apr 21 10:04:16 EDT 1992
  *** Starting crev and twk_put *** Tue Apr 21 10:04:18 EDT 1992
yes to proceed, CTRL/C to abort: yes
crev and twk_put pass for ANALYZE Tue Apr 21 10:04:21 EDT 1992
crev and twk_put pass for COLLECT Tue Apr 21 10:06:12 EDT 1992
  *** Completed crev and twk_put *** Tue Apr 21 10:10:46 EDT 1992
End RET Build Program, Tue Apr 21 10:10:47 EDT 1992
```

If the system had been built once already but changes had occurred only in ANALYZE, the user could rebuild only that CSCI by issuing the same set of commands, but with ANALYZE appended to each.

Options 6 and 7 are not a normal part of building the system. They are useful for backups and for transporting the database between systems. They simply invoke the appropriate Cadre utilities. When option 6 is invoked without a CSCI name, the dump is placed into $RET_DB_HOME/dump/twk-dump. If a CSCI name is specified, then the dump goes into $RET_DB_HOME/dump/*csci_name*.twk-dump. When option 7 is chosen, it loads the files from the dump files written in option 6.

Build-ret also produces log files. These are found in the directory $RET_DB_HOME/log. Here is a list of the log files and where they are produced:

pass1	*csci_name*.p3-log	
pass3	*csci_name*.p4a-log, p4b-log, *csci_name*.p4c-log	
pass5	*csci_name*.twk-log	
pass6	twk-dump-log	(if invoked without CSCI name)
	csci-name.twk-dump-log	(if invoked with CSCI name)
pass7	twk-load-log	(if invoked without CSCI name)
	csci-name.twk-load-log	(if invoked with CSCI name)

2.2 RET Installation

Once the distribution tape is received, the contents should be extracted using **tar** (a Unix utility). This will create a directory named **ret**, with several subdirectories. All RET users will need to create an environment variable, **$RET_DB_HOME**, which contains the path name of this ret directory.

There are several files in the directory $RET_DB_HOME/sys which must be customized. In the files listed below, the string "$RET_DB_HOME" must be replaced with the hard-coded path name of your installation's ret directory (e.g. /common/sun4/ret). The affected files are:

dd.menu	(1 substitution)
dde.menu	(1 substitution)
desktop.menu	(1 substitution)
dpi.menu	(1 substitution)
file.menu	(1 substitution)
ms.menu	(1 substitution)
pi.menu	(2 substitutions)
sc.menu	(4 substitutions)
config_file	(10 substitutions)

The only other requirements are that the Cadre Team*work* and Crev products must be installed. Please refer to the Cadre documentation [4] for this procedure.

2.3 System Setup

Once RET has been installed, the user must load into it the source system to be examined. There are two steps involved with this:

1. in $RET_DB_HOME/src, update the file **search.paths** to contain only the names of the CSCI's which are part of the system to be examined.
2. in $RET_DB_HOME/src, create a soft UNIX link to each CSCI entered in **search.paths**. (Each CSCI should now have a directory
3. filled with the source files associated with it.)

It should be noted that CMS RET views a CMS-2 system as a set of CSCIs. Each CSCI is a subdirectory of the overall system directory, containing source files which are presumably related. Even if there is only one source directory for a project, it should appear as a subdirectory of the project itself, and be considered a CSCI. It is generally advisable for the CSCIs' names to be all capital letters.

Chapter 3.0 Information Extraction

3.1 User Perspective

The user will run this pass on every complete source file in the CMS-2 system. (*Include* files are brought in automatically by the files which reference them.) This can be done using the build-ret script described in Section 2.1, or by issuing the command:

cms2cdif.p3 -csci ***csci_name { file_names }***

so that each source file in every CSCI is processed. *file_name* is a non-empty list of the files to be processed and *CSCI_name* is the name of the CSCI containing these files. (*csci_name* must not contain wild cards, but *file_name* may.) The command must be issued within the appropriate CSCI. For each file processed, there will result a middle file and a comment file, which are used in the later steps. The formats of the middle and comment files are given in Chapter 8.0.

3.2 Internals

The Information Extractor is written in Ada, and has two basic parts (the parser and the extractor), both of which interface to our internal representation of the CMS-2 language. The parser and internal representation were completely reused from the CMS2Ada translator, with a few extensions to expand our language coverage. (These are detailed in the description of the parsing package in 3.2.2.) Parts of the extraction mechanism were adapted from JRET (the Jovial Reverse Engineering Technology), but much of it was rewritten because of the differences between the internal representations of CMS-2 and Jovial. The new version was written with liberal use of generics and non-language specific data structures, with the hope that most of it will be reusable should we ever want to reverse engineer another language.

The remainder of this section contains a brief description of the packages in the Information Extractor, the relationships between them, and a more detailed look at some of the more important packages.

3.2.1 The packages comprising the Information Extractor

In the list that follows, * indicates almost complete reuse, # indicates significant reuse, and *italics* indicate that the package is generic.

- Main * (contains main driver, handling command-line interface and file control)
- CMS_Records * (description of the nodes which make up the internal representation)
- CMS_Interface * (access routines for CMS_Records)
- CMS_Utils (some general utilities not available in CMS_Interface)
- Parse * (creates a parse tree, made up of structures from CMS_Records)
- Lexical Analysis *

- Symbol tables and symbol table management * (several related packages)
- *Parse Control* * (helper package for parser and classification routines)
- Extract_Info # (high-level node-processing routines; basically sorts out the nodes)
- Data_Processing # (mid- and low-level routines specific to data declarations)
- Executable_Processing # (mid- and low-level routines specific to executable statements)
- Option_Processing (mid- and low-level routines specific to option statements)
- Structure_Processing (mid- and low-level routines specific to structure statements)
- Subprogram_Processing # (mid- and low-level routines specific to subprogram declarations)
- *Print_Middle* # (language-independent printing routines; mainly utilities)
- *Source_file_database* * (associates nodes with file names and line numbers)
- Scoping# (determines which data items are global and which are local)
- *Subprogram_Lists* # (data package for communication between Subprogram_Processing and Executable_Processing)
- System_Info (data package indicating what options are currently in effect and what structure is being processed)
- Debug_Flags * (framework for dubugging)
- *Comment_Handler* # (received comments and context indications from the parser and prints to the comment file as appropriate)
- Comment_Helper (Language specific utilities unique to comment handling)

Figure 5 shows the with'ing relationships between these packages.

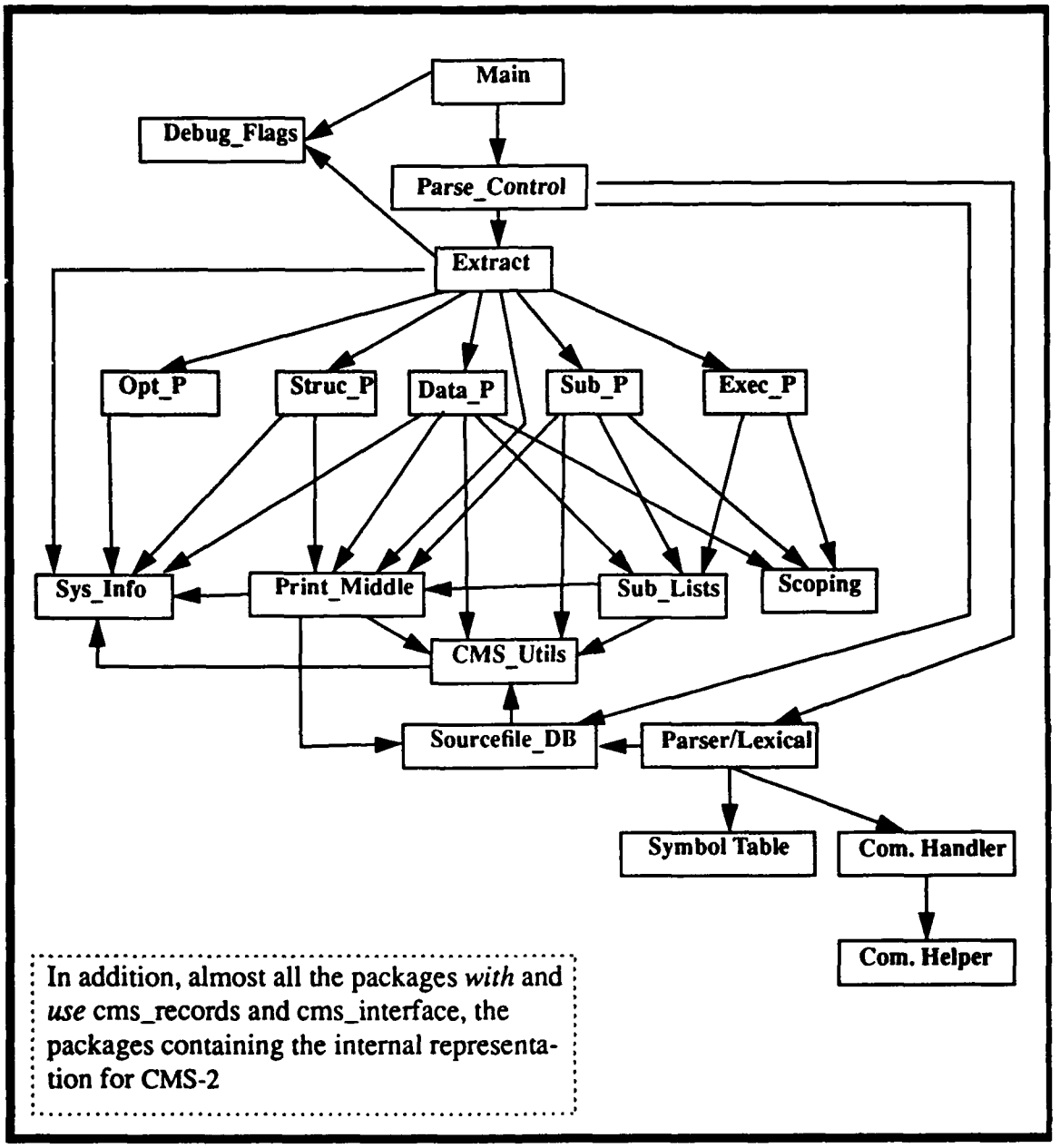


FIGURE 5. With'ing Relationships between Packages

3.2.2 Details of Important Packages

Parsing and Representation Packages: These include *CMS*_Records, CMS_Interface, and the parser, lexical analyzer, and symbol-table packages. As a baseline, we reused these packages from the CMS2Ada translator, but extended them as part of this contract to address certain CMS-2 constructs which were not previously handled. These include macro expansions (via the *means* and *exchange* statements), user-defined type declarations, and the *terminate* phrase. In addition, a means for processing *cswitch* directives was designed, but not implemented.

Parse_Control: is a generic package containing a pointer to the top node of the parse tree and the routine which calls the (instantiated) parser and extractor.

Extract_Info: contains the top-level extraction routine, and those generalized routines which classify each node and drive the processing. The top-level extraction routine also takes care of the file control for the middle file being created.

Visible Routines:

Process_A_Node
Process_Seq_Of_Nodes
Is_Receptacle
Process_Receptacle
Process_Seq_Of_Receptacles
Is_Expression
Process_Expression
Process_Seq_Of_Expressions
Extract

Data_Processing: contains routines to classify and process the nodes which represent data declarations. The processing includes checking the declaration for usages of other data items, and printing appropriate information to the middle file.

Visible Routines:

Is_Data_Decl
Process_Data_Decl

Executable_Processing: contains routines to classify and process the nodes which represent executable statements. The processing includes checking for data uses and subroutine calls, and keeping track of any which are found.

Visible Routines:

Is_Executable_Node
Process_Executable_Node

Option_Processing: contains routines to classify and process the nodes which represent option statements. The processing generally entails setting global variables to reflect the options found.

Visible Routines:

Is_Option_Node

Process_Option_Node

Structure_Processing: contains routines to classify and process the nodes which represent structural statements. This processing generally consists of making sure all statements within the structure are processed.

Visible Routines:

Is_Structure_Node
Process_Structure_Node

Subprogram_Processing: contains routines to classify and process the nodes which represent subprogram declarations. This processing includes setting up a framework in which to collect information about the subprogram's activities, making sure all statements within the subprogram are processed, and writing the information collected to the current middle file.

Visible Routines:

Is_Subprogram_Decl
Process_Subprogram_Decl

Print_Middle: is a generic package which contains a file pointer to the middle file, and routines to handle much of the printing for it. The idea behind this package is that the format of the middle file is language-independent, even though the internal representation of the information is not. Therefore, the routines in print_middle use language-specific instantiated "helper" routines in order to access any extra information needed, and then print everything out in a standard format.

Visible Routines:

New_Middle_File
Get_Middle_File
Close_Middle_File
Comma_Space
Print_Component_Decl
Print_Extended_Name
Print_Formals
Print_Simple_Decl
Print_Start_of_Composite_Decl
Print_Source_Info
Print_TW_Attribute

Subprogram_Lists: is a generic package which contains the infrastructure which the subprogram_processing routines use to keep track of the reference information collected. It serves as the prime communication mechanism between the Subprogram_Processing and Executable_Processing packages.

Visible Routines:

Add_To_Calls
Print_Calls
Add_To_Reads
Print_Reads

Add_To_Writes
Print_Writes
Print_Reads_And_Writes
Initialize_Subprogram_Lists
Update_Reads_And_Writes
Push_Locals
Pop_Locals
Add_To_Locals
Add_To_Params
Mem_Locals
Mem_Params

Chapter 4.0 Comment Processing

4.1 User Perspective

The user will run this pass on every **.comment** file produced as a result of the Information Extraction. This can be done using the build-ret script described in Section 2.1, or by issuing the command

gawk -f $RET_DB_HOME/cmsrev/bin/coments.awk *.comments

(see Section 2.2 for the proper setting of the $RET_DB_HOME environment variable). (*gawk* is gnu awk. If your installation does not own a copy, use the one in $RET_DB_-HOME/cmsrev/bin.) Since this capability must be sensitive to the commenting conventions of the current project, it is recommended that the user customize the **comments.awk** program to reflect the prevailing conventions. Those planning to do this customization would be well-advised to read Section 8.2, which describes the format of the **.comment** files.

4.2 Internals

The **.comment** files written by the Information Extractor contain a line for each comment found, and one for each "interesting" construct encountered in the source code. Interesting constructs include data declarations, subprogram declarations, header blocks, *proc* and *dd* statements. Thus the files contain not only the comments, but some context condensed out of the source code. A distinction is made between *COMMENT ... $* constructs and in-line comments, resulting in even more context information.

The purpose of the **comments.awk** program is to create an **.ext_com** file for each subprogram declaration found in a **.comment** file. This **.ext_com** file contains exactly the text that will eventually appear in the Mspec for that subprogram in the Cadre database. The standard **comments.awk** program, included with this release, selects as relevant the comments which fall between the subprogram's declaration and its actual code.

Chapter 5.0 System Integration

5.1 User Perspective

There are several passes involved in this activity. They can be run via the build-ret script described in Section 2.1, or by issuing the following commands:

(in each CSCI directory)
cms2cdif.p4a -csci *csci_name* *.middle
(in parent directory)
cat *.decls | sort | awk -f $RET_DB_HOME/admin/p4b.awk
cms2cdif.p4b -P search.paths *.export
(in each CSCI directory)
cms2cdif.p4c -csci *csci_name* -crev -mspec -dde *.middle
$RET_DB_HOME/admin/do_post *csci_name*

(See Section 2.2 for a description of the $RET_DB_HOME environment variable.) If the passes are run outside of the build-ret script, there is a set of files which must exist before running them. In each CSCI, **limits.txt** must be present. This should be copied from $RET_DB_HOME/admin, or it can be made an empty file, in which case no DDE's will be produced. In the CSCI's parent directory, **search.paths** must exist. It will contain the names of the CSCI's which are to be active (this would typically be all of the subdirectories).

The output of these steps is the set of files **twk.script**, **ret.crev**, **ret.ctl**, **ret.dd** and **ret.ms**. These are used in building the Teamwork/SD reverse engineering database.

5.2 Internals

The purpose of these steps is to reconcile any name clashes which may occur, either within or between CSCI's, to resolve inter-CSCI references, and to build the CDIF[1] representation of each CSCI's information. Briefly, the processing responsibilities are divided as follows: **cms2cdif.p4a** compiles two lists for each CSCI, one for data item names and one for subprogram names. **p4b.awk** and **cms2cdif.p4b** create new names where necessary to avoid name clashes. **cms2cdif.p4c** creates the CDIF files which will be fed into the Teamwork database, and a script for loading them. **do-post** edits a few files so that the Teamwork extensions will read them correctly.

1. CASE Data Interchange Format

Chapter 6.0 Building the Teamwork/SD Reverse Engineering Database

6.1 User Perspective

The CSCI's for the databases being constructed must exist in Team*work*. If they do not, then start Team*work* and create new models with these CSCIs' names. Once the models exist, construct their respective databases either by using the build-ret script described in Section 2.1, or by issuing the following command in each CSCI:

/bin/sh twk.script

6.2 Internals

This step invokes **crev** and **twk_put** to build the database. **crev** uses **ret.crev** and **ret.ctl** to produce the Team*work* structure charts, and **twk_put** creates Mspecs from **ret.ms**, and DDEs from **ret.dd**.

Chapter 7.0 TeamWork Environment

7.1 Invocation

In order to use the extensions GE-supplied extensions, Team*work* must be invoked using the RET config_file. This config_file must be customized during installation, as described in Section 2.2. Once that is done, invoke Team*work* as follows:

teamwork -c $RET_DB_HOME/sys/config_file

(See Section 2.2 for the proper setting of the $RET_DB_HOME environment variable.)

7.2 Basic Team*work* Displays

Most of the Team*work* displays are standard to the Team*work* environment, and are explained in the Cadre documentation. The Mspec and DDE displays are somewhat customized for RET, so they are described here.

The Mspec (Module Specification) display is intended to describe the important aspects of a module. In this context, a module corresponds to a subprogram. The information contained is the following: subprogram parameter names and directions; global variables accessed, along with an indication of whether they are read or written; modules called; calling modules; and comments extracted from the source code of the module.

The DDE (Data Dictionary Entry) display is intended to convey the important features of a data item. The information supplied for a simple variable includes: type information; actual location (file and line number) of its declaration; and location of its declaration, taking into account *include* expansions. For arrays, the number of dimensions, direction, and any field names are also included.

7.3 GE-Supplied Extensions

The user should consult the Cadre documentation for information on the standard Team*work* environment [3]. What follows here is a description of the GE-supplied extensions to that environment, and guidelines for how to use them.

Displaying Source Files: There are times when the summarized information is not sufficient for the task at hand. In these cases, it is useful to have a quick method of viewing the actual source code. In order to do this, select a module of interest from a structure chart or Mspec, or a data item from a DDE, and choose the RET menu item "Display Module Source". The corresponding source file will be displayed, and the user can then search on the name of the module or data item in order to find the desired declaration.

Displaying Data Usages: It is often important to know which modules use a particular global variable. This information is available from the full Data Dictionary as well as the Mspec display. To view it, simply select the desired global variable, and choose either "Display Where Ref" or "Display Where Ref All" from the RET menu (the latter extends the search across all active CSCIs). The information will be retrieved and displayed in a window which lists the modules in which that data item is referenced. From that window, the user may move to the Mspec for any of the referencing modules by selecting its entry and choosing the RET menu item "Show Module Spec".

Displaying Calling Modules: Although the structure charts are effective in showing the called modules of a particular subprogram, it can be tedious working backwards to find the calling modules. There are two ways to find this information easily. The first method is to view the Mspec of the desired module and find the list of calling modules. The second method is to select the desired module from a structure chart and choose the RET menu item "Display Calling Modules". The information will be retrieved and displayed in a window which lists the modules which call the selected one. From that window, the user may access the Mspec for any calling module by selecting its entry and choosing the RET menu item "Show Module Spec". (From there, "Show SC" from the Whole_Mspec menu will bring up the corresponding structure chart.)

Displaying Mspecs from Structure Charts: When viewing a structure chart, select the desired module and choose the RET menu item "Open Module Spec". The corresponding Mspec will appear.

Displaying DDE's for the Fields of a Table: When viewing the DDE of a table or array, it is not enough to see just that item's information; the component items' entries are equally important. These can be viewed easily by highlighting the desired name within the table's DDE and then choosing the RET menu item "Open DDE". A new DDE window will open with the desired entry.

For a more in-depth description of the GE-enhanced Team*work* environment, please see the *CMS RET User's Manual* found in Appendix A.

Chapter 8.0 File Formats

8.1 Middle Files

The middle files hold the information which is extracted from the CMS-2 source files, before it is integrated into a system view. In the case that this technology were ported to a CASE tool other than Cadre, these files would be the starting place for the re-implementation.The following is the grammar for the middle files.

file ::= "file" string_literal ["csci" identifier] {declaration }

declaration ::= context_decl | external_decl | subroutine_decl | object_decl |
group_decl | type_decl

context_decl ::= "context" identifier [id_list] [source_info]

context_list ::= { context_decl }

external_decl ::= "external " global_declaration

global_declaration ::= subroutine_decl | object_decl | type_decl

subroutine_decl ::= procedure_decl | function_decl

procedure_decl ::= "procedure" identifier [source_info]
subroutine_info "end"

function_decl ::= "function" identifier {source_info] type_info
subroutine_info "end"

subroutine_info ::= ["long" "name" string_literal]
[formal_list] [local_list] [context_list] [calls_list] [reads_list]
[writes_list] [reads_writes_list] [nested_subs_list]
["header" "file" string_literal] ["copy" "files" string literal]
[pseudo_code_list]

object_decl ::= simple_decl | composite_decl

simple_decl :: = "simple" identifier ["constant"] [source_info]
["csci" identifier] tw_attr type_info ["members" list]

composite_decl ::= "composite" identifier ["constant"] composite_class
[source_info] ["csci" identifier] [index_info] tw_attr
(component_list | type_info)

index_info ::= "indexed" "(" integer_literal ")"

```
tw_attr ::= [ tw_prim ] tw_flow

tw_prim ::= "PEL" | "CEL" | "DEL"

tw_flow ::= "controlflow" | "dataflow" | "bothflow" | "store"

group_decl ::= "group" identifier [ source_info ] [ "csci" identifier ]
        { declaration } "end"

formal_list ::= "formals" "(" formal { "," formal } ")"

formal ::= identifier direction type_info

local_list ::= "locals" id_list

a_call ::= identifier [ "nested" ] [ actual_list ]

actual_list ::= "(" actual { "," actual } ")"

actual ::= ( "(" object_decl ")" ) | identifier

direction ::= ( "in" [ "out" ] ) | "out"

type_decl ::= "type" (simple_type_decl | composite_type_decl )

simple_type_decl ::= "simple" identifier [ source_info ] [ "csci" identifier ]
        tw_attr type_info [ "members" list ]

composite_type_decl ::= "composite" identifier composite_class
        [ source_info ] [ "csci" identifier ] [ index_info ] tw_attr
        ( component_list | type_info [ "members" list ] )

component_list ::= "(" [ ( simple_decl | composite_decl )
        { "," ( simple_decl | composite_decl ) } ] ")"

composite_class ::= string_literal

type _info ::= string_literal

calls_list ::= "calls" "(" a_call { "," a_call } ")"

reads_list ::= "reads" id_list

writes_list ::= "writes" id_list

reads_writes_list ::= "reads_writes" id_list
```

nested_subs_list ::= "nested" { subroutine_decl }

pseudo_code_list ::= "pseudo_code" list

list ::= "(" string_literal { "," string_literal } ")"

id_list ::= "(" identifier { "," identifier } ")"

source_info ::= integer_literal string_literal [integer_literal string_literal]

identifier ::= string_literal | reserved_word_of_language

8.2 Comment Files

The comment files contain both the CMS-2 comments and some condensed context information. These are the files which are input to the comment processor, which then produces one **.ext_com** file for each subprogram, containing any relevant comments. The awk script of the comment processor is user-customizable.

file ::= {entry }

entry ::= comment_entry | context_entry

comment_entry ::= same_line_entry | stand_alone_entry

same_line_entry ::= "SAME LINE: " string

stand_alone_entry ::= "COMMENT: " string

context_entry ::= data_decl | subprogram_decl | structural_entry |
"CODE" | "UNKNOWN CODE"

data_decl ::= "DATA" | "EQUALS" | "FIELD" | "LOADVRBL" |
"NITEMS" | "PARAMETER" | "SYS-INDEX" | "TABLE" |
"VARIABLE"

subprogram_decl ::= "EXEC-PROC" identifier |
"FUNCTION" identifier |
"PROCEDURE" identifier |
"END"

structural_entry ::= "AUTO-DD" | "END-LOC-DD" |
"END-MAJOR-HEADER" | "END-SYS-DD" | "END-SYS-PROC" |
"END-SYSTEM" | "LOC-DD" | "MAJOR-HEADER" |
"MINOR-DD" | "PROGRAM-BODY" | "SUBPROGRAM-DD" |
" SYS-DD" | "SYS-PROC" | "SYSTEM"

Part II ENCORE-SRE Integration Study

Task III of this project sought to study the feasibility of integrating GE's ENCORE system with Computer Command and Control Corporation's (CCCC) Software Re-Engineering Environment (SRE). (The original title of this contract refers to MODEL which was developed at CCCC as part of their reengineering efforts, but the reengineering environment to which this study refers is SRE.) The initial phase of the study compared the functionality of the two systems to determine whether it makes sense to integrate them. This was followed with the design of a method for integrating the two systems. As a result of our study, we have concluded that the two systems could functionally complement each other and that there are no insurmountable technical barriers blocking the integration. The issues involved with integrating the two systems are discussed in the following paragraphs.

The ENCORE system promotes reuse of heritage code via automatic translation and reengineering. Components of the ENCORE system include translators from FORTRAN to Ada and CMS-2 to Ada, control and data restructuring, basic metric capabilities, limited dataflow analysis, and the ability to parse and regenerate Ada programs. The restructuring components (control and data) provide an automated mechanism for understanding and improving the fine grained aspects of a software system. The SRE system provides an environment for viewing and modifying the coarse grained architectural features of an existing software system. Combining ENCORE and SRE would produce an environment for reengineering both at the fine grained and coarse grained levels.

Combining the two systems would require that they share the information about the code being reengineered. Currently both systems operate on their own distinctive internal representation of Ada code. (The ENCORE internal representation is called the IRep and the SRE internal representation is called the ESL.) Since the implementations of the two representations are vastly different and a great deal of reengineering functionality has already been developed specific to each implementation, we recommend a loose coupling of the two systems via translation between the two internal representations. Though the implementations of the two internal representations are vastly different, they both embody the same information and the mapping from one internal form to the other appears to be straightforward. (The IREP is described in Appendices B, C, and D, while the ESL is presented in Appendix E.)

This approach avoids the reimplementation of reengineering capabilities just for a different internal representation and it allows the two companies to further develop their products without having to tightly coordinate changes.

The only stumbling point in this integration scheme is a platform problem. The ENCORE system runs on a UNIX[1] platform and currently uses the SunView[2] windowing system. The SRE system is tightly coupled with the DECdesign[3] system and therefore must run on a

1. UNIX is a registered trademark of AT&T Bell Laboratories
2. SunView is a trademark of SUN Microsystems, Inc.
3. DECdesign is a trademark of Digital Equipment Corporation

VMS platform. This problem can be overcome by either moving one system to the other platform, or creating a mechanism for passing the information between the internal representations (and therefore between machines) via ASCII files.

If the ENCORE user interface were rewritten in X, then ENCORE could run on the VMS platform. To move SRE to a UNIX platform, CCCC would have to either get an implementation of DECdesign for UNIX or replace the use of DECdesign in SRE with some other database and visualization system. Either option involving SRE is estimated to require more effort than moving ENCORE to VMS. We advocate changing the ENCORE user interface to X, if integrated performance on a single platform is required.

The alternative to changing platforms is to provide a mechanism for passing the information between the two reengineering systems via ASCII files. To use the systems in an integrated manner, one would follow the following sequence of steps: 1) a collection of Ada source code would be reengineered using one of the systems; 2) ASCII files capturing the all the necessary information would be generated and passed to the other system; 3) the other system would be used to further reengineer the Ada. (The passing of the ASCII files would be bi-directional.) When the reengineering is finished, new Ada code would be regenerated capturing the reengineering modifications made by both systems. With this scenario, the ASCII files would have to completely capture all the information in the internal representations and both systems would have to be able to parse and print these ASCII files.

Since the internal representations for the two systems currently contain the exact same information as is contained in an Ada program, we have the option of choosing either an abstraction of one of the internal forms or restructured Ada code for the format of the ASCII files. A shortfall of the latter option is that it precludes future expansion of the internal representations. We expect that in the future we will want to expand SRE and ENCORE to be able share computed information about the Ada code. Using Ada code as the means of communication between the two systems would prohibit this expansion. Therefore we recommend choosing an abstraction of one of the internal forms.

The ENCORE system currently has a prototype version of an ASCII file parser and printer which consumes and produces an abstraction of the IRep. We call this software our IRep Inputter/Outputter. As mentioned above, this software is only in prototype form at this time, but with minimal effort it can be extended to handle the complete ENCORE IRep. The software is written in Ada and can easily be integrated into both SRE (under VMS) and ENCORE (under UNIX).

In summary, we are suggesting translation between the SRE ESL and ENCORE IRep as the best way to integrate the two systems. To accomplish this an ESL $\leftrightarrow$ IRep translator must be built and either ENCORE will have to be moved to the VMS platform, or the IRep Inputter/Outputter will have to be made more robust and incorporated into both SRE and ENCORE. The following pages provide an outline and estimates of the tasks involved with each option.

Figure 6 illustrates the envisioned system architecture for a merged ENCORE-SRE system where ENCORE has been moved to the VMS platform.

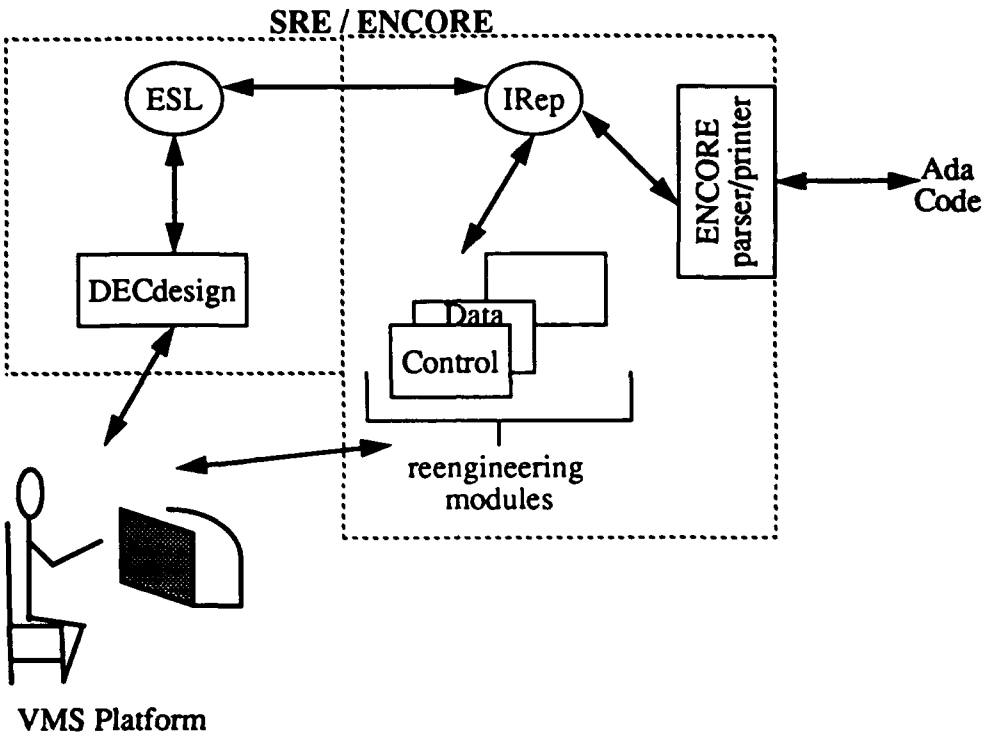


FIGURE 6. ENCORE - SRE Integration on a Common VMS Platform

To realize the integration shown in Figure 6, the following tasks must be completed:

- Software must be written to translate back and forth between ESL and IRep. (about 9 person months to complete)
- The ENCORE user interface must be rewritten in X. (about 9 person months)
- The ENCORE and SRE user interfaces must be updated to allow the user to switch between the two systems (automatically transferring from one internal representation to the other). (about 1 person months - 1/2 person month for each system)

We believe a sound estimate for this form of integration is 20 person months.

Figure 7 shows how to integrate the two systems via ASCII IRep files.

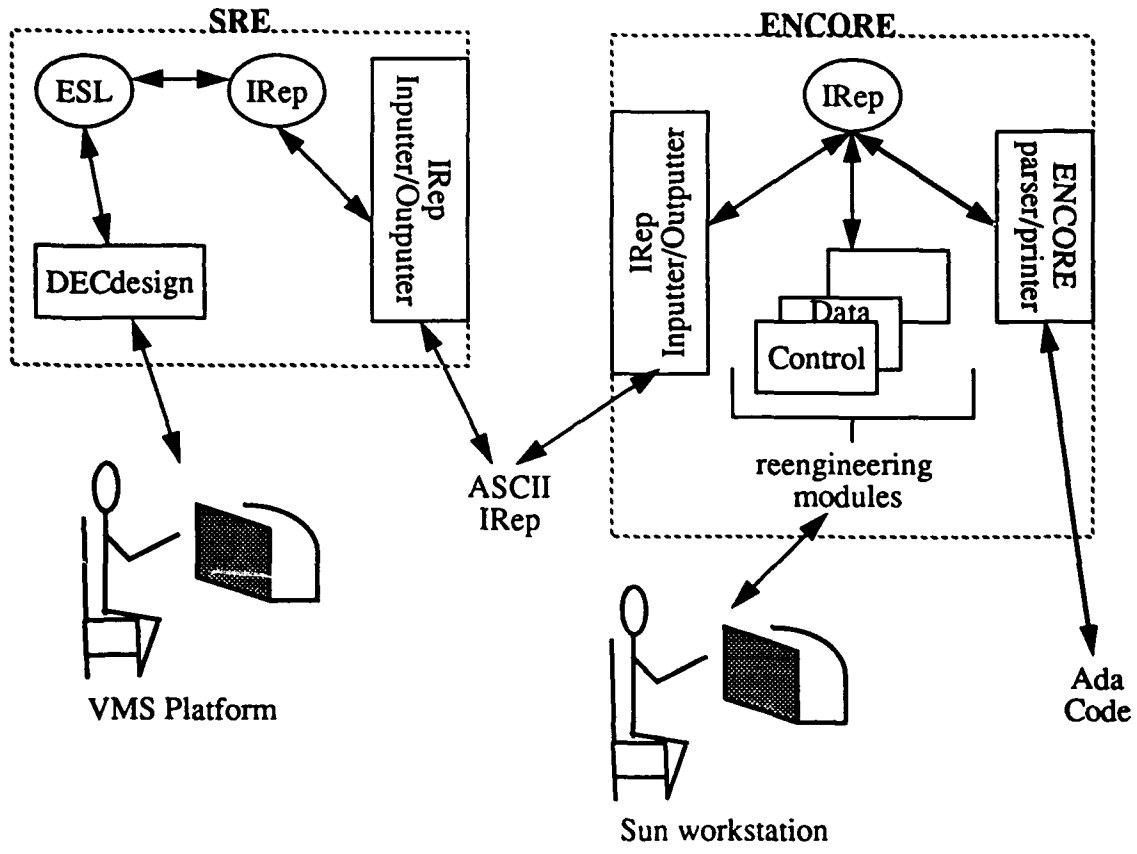


FIGURE 7. ENCORE - SRE Integration via ASCII IRep Files b

To implement the above integration the following must be done:

- Software must be written to translate back and forth between ESL and IRep. (about 9 person months to complete) (This is the same as the first bullet with the previous integration option.)
- The IRep Inputter/Outputter must be made more robust. (about 3 person months to complete)
- The IRep Inputter/Outputter must be incorporated into the SRE system. the file loading process must be updated to load IRep files using the Inputter and translate the IRep structure to an ESL structure, and the file writing process must be updated to translate the ESL structure to the IRep structure and generate the ASCII IRep files using the Outputter. (about 2 person months to compete)
- The ENCORE file load and file write must be updated to use the IRep Inputter/Outputter. (about 1 month to complete)

We estimate it will take 16 person months to achieve this form of integration.

References

[1] CMS2Ada - a CMS-2 to Ada translator developed at GE Corporate Research and Development. For information contact J. Sturman at GE Corporate Research and Development, P.O. Box 8, Schenectady, N. Y. 12301 (518) 387-5457

[2] JRET - Jovial Reverse Engineering Technology developed at GE Corporate Research and Development. For information contact J. Sturman at GE Corporate Research and Development, P.O. Box 8, Schenectady, N. Y. 12301 (518) 387-5457

[3] *Teamwork/SD User's Guide, Release 4.0*, Cadre Technologies, Inc., 1990

[4] *Teamwork System Administrator's Manual, Release 4.0,* Cadre Technologies, Inc., 1991

Appendix A

CMS RET User's Manual

CMS RET User's Manual

1.0 Introduction

The CMS Reverse Engineering Tool (RET) consists of CMS Rev and Teamwork/SD. CMS Rev has been developed by GE CR&D and provides the ability to process CMS-2 source code[1] and create a software maintenance database. This software maintenance database consists of a Teamwork database of structure charts, module specs, data dictionary entries and collateral files which contain information about the structure and contents of the source code being maintained. Teamwork/SD is a commercially supported product available from Cadre Technologies. It has been augmented by user menus, shell scripts and access programs to provide a customized and enhanced environment which utilizes the software maintenance database created by CMS Rev.

This CMS RET User's Manual describes the procedures for using the RET software maintenance database. These procedures involve the use of the Teamwork/SD product from Cadre Technologies. The section ***Using the RET Database*** documents the basic operations that the software maintainer would need to perform in order to utilize the software maintenance database.

Also contained in this manual are the procedures for creating the RET software maintenance database using CMS Rev, related programs and shell scripts. These procedures are performed in batch mode when necessary because of a new release of the source code being maintained, or as a result of new version of CMS Rev. The section ***Building the RET Database*** documents the steps necessary to build a new RET software maintenance database. The section ***Installing the RET Processors*** documents the steps necessary to install CMS Rev, related programs and shell scripts before beginning the process of building a new RET database.

2.0 Using the RET Database

2.1 Invoking RET

RET is invoked by executing Teamwork using the RET configuration file. This can be accomplished by manually typing the Teamwork command or by selecting the appropriate menu item from an OpenWindows workspace menu. The user then interacts with Teamwork to access the RET software maintenance database. The RET configuration file provides the user with access to the customized RET menus and to the specialized programs which access the RET software maintenance database.

A number of setup operations need to be performed before RET can be invoked: (1) Modify the Unix PATH variable to include the Teamwork directories. (2) Initialize the Unix environment variable RET_DB_HOME to specify the RET root directory. (3) Verify that the Teamwork DC server is running on the Teamwork workstation server.

2.2 Using the Online Help

Each of the RET menus has a menu selection titled "Display Help Screen" that is the last selection on the menu. Selecting this menu item will cause the context sensitive help screen to be displayed in a Teamwork

1. The current prototype handles a subset of the dialect known as CMS-2L. This subset is detailed in *Detailed Design for CMS-2 to Ada Translation*, delivered to NSWC in January 1992.

window. In addition to a description of each menu item available to the user, there may appear hints to the user on how to perform specific operations.

2.3 Selecting the Model of Interest

The first operation that the user must perform is to select the model of interest. Any further operations will then pertain to is model which corresponds to a Unix directory.

The model of interest is selected by pulling down the Index menu from the desktop menubar, and selected the menu item titled "Open Model Index." This will cause a list of the models in the Teamwork database to be displayed. Highlight the model of interest and select "Open PI" from the pullright menu. The Teamwork process index for the selected model will be displayed. From this process index window, the user may access structure charts, module specs, and the data dictionary associated with the selected model.

2.4 Navigating Structure Charts

Structure charts provide a graphical representation of the calling relationships between software modules. The structure chart can be used as a "map" to guide the software maintainer in his/her understanding of the underlying software. Off-page connectors are used in structure charts so that the amount of information on a given structure chart is not excessive. The intent is to maintain readability when RET-generated structure charts are printed on 8 1/2 by 11 inch pages.

To navigate downward in the module calling hierarchy using structure charts, the user may open the structure charts for a specified off-page connector. This is accomplished by selecting the off-page connector with the mouse select button (left mouse button), pulling down the RET menu from the structure chart menubar and selecting the menu item titled "Expand Connector." The structure chart for the off-page connector will then be displayed.

To navigate upward in the module calling hierarchy using structure charts, the user may request to display a list of modules which call the current module. The current module is, by default, the module at the top of the structure chart. The user may override this default module by explicitly selecting another module on the structure chart as current. The list of calling modules is obtained by pulling down the RET menu from the structure chart menubar, and selecting the menu item titled "Display Calling Modules." This will cause a file window to open with a listing of calling modules. Any line of this file display may be selected to request the structure chart for that calling module by pulling down the RET menu from the file menubar and selecting the menu item titled "Open Structure Chart." The structure chart for the selected calling module will then be displayed. (NB: this is not currently working. Workaround: choose "Open Module Spec" from the RET menu and then choose "Show SC" from the Whole_Mspec menu).

During the search, an icon is displayed with the title "CALL." This icon will disappear when the search is completed, and at that time a Teamwork window will display the results of the search. The "Display Calling Modules" request may be aborted using the normal Unix window procedure to quit a task represented by the CALL icon.

2.5 Selecting the Module of Interest

A module is a CMS-2 subprogram. Modules are identified by the RET and a module spec is created for each module. In addition, the boxes on structure charts are used to represent modules.

Modules are listed on the process index which is displayed when the model of interest is selected. The process index lists the module specs and structure charts that are contained in the Teamwork database. The module of interest may be selected from the process index, and then either a module spec or a structure chart

may be opened. Each process index entry has a SC or MS indicated. SC refers to structure chart and MS refers to module spec. A structure chart may be opened by selecting a module name flagged with an SC, pulling down RET from the process index menubar and selecting the menu item titled "Open Structure Chart." A module spec may be opened by selecting a module name flagged with an MS, pulling down RET from the process index menubar and selecting the menu item titled "Open Module Spec."

Structure charts show the calling relationships between modules. The module of interest may be selected from a structure chart by pointing the mouse cursor at the structure chart that represents the module of interest, and pressing the select mouse button (left mouse button). Then, the user may pull down the RET menu from the structure chart menubar and select the desired menu item. The module of interest may be selected from a module spec by selecting the text in the module spec body which is the name of the module of interest. Text in the module spec body is selected by moving the mouse cursor on top of the first letter in the text string. The mouse cursor should turn from an arrow into a block. The select mouse button (left mouse button) is then pressed and the mouse cursor is dragged across the letters of the text string. The selected text will appear in reverse video. Then, the user may pull down the RET menu from the module spec menubar and select the desired menu item.

2.6 Determining Module Interfaces

A module interface is a relationship between modules where one module calls the other module. Module interfaces are represented graphically by structure charts, and textually by information in module spec bodies. Module interfaces may be obtained by displaying the appropriate structure chart or module spec. From a module spec, the user may open a Teamwork window for the structure chart containing the module spec. This is accomplished by pulling down the Whole_Mspec menu from the module spec menubar, and selecting the menu item titled "Show SC." This will cause the appropriate structure chart to be displayed.

2.7 Viewing Module Source Files

Module source files are the raw CMS-2 source files. Module source files may be displayed by selecting the module of interest from a structure chart, or from a module spec body. Then, the RET menu item titled "Display Module Source" may be selected to complete the request for a Teamwork file window to be opened on the raw source file.

An important distinction to remember is that the name of a module is not necessarily the same as the name of the source file containing the module. The boxes on structure charts and the "calls" and "called by" section of the module spec body all use module names, not file names.

2.8 Searching Source Files for Text

A facility for searching raw source files has been built into RET. This facility is available from the process index menubar. The user selects the model of interest and opens the appropriate Teamwork process index window. The user then pulls down the RET menu from the process index menubar, and selects the menu item titled "Search Source Files." This causes a Teamwork input window to be displayed which requests the user to input the filename and text patterns. The filename pattern is a standard Unix filename pattern, including the use of ? and * for wildcards. The text pattern is a grep regular expression, which needs to be enclosed within either single or double quotes if the text pattern contains special characters.

After the filename and text patterns are input, a Unix task is invoked to perform the search on the source files. During the search, an icon is displayed with the title "SCH." This icon will disappear when the search is completed, and that time a Teamwork window will display the results of the search. The "Search Source Files" request may be aborted using the normal Unix window procedure to quit a task represented by the SCH icon.

Source files may also be searched across all CSCIs. This is accomplished using the "Search Source Files" menu item on the RET menu of the Teamwork desktop menubar.

2.9 Selecting the Global Variable of Interest

Global variables are variables which are used outside of the module in which they are declared. These global variables are listed alphabetically within the data dictionary for each model, and also as part of the module spec for modules which reference the global variable. The data dictionary is displayed for the model of interest by pulling down the "Whole_Model" menu from the process index menubar, and selecting the menu item titled "Open DD." This will cause the requested data dictionary to be displayed. The global variable of interest may be selected from this display of the data dictionary by moving the mouse cursor to the desired line of the data dictionary and pressing the select mouse button (left mouse button).

When a module spec is displayed, the global variables listed may also be selected as the global variable of interest. This is accomplished by moving the mouse cursor on top of the first character of the name of the global variable. The mouse cursor will change from an arrow to a block. The user presses the mouse select button (left mouse button) and drags the cursor across the global variable name until all the characters are in reverse video. At this point, the global variable of interest on the module spec has been selected.

2.10 Viewing Data Dictionary Definition of Global Variables

The data dictionary contains an entry for each global variable. This entry contains information about the global variable, including the actual declaration of the global variable, and information about the raw or expanded source files. The declaration contains the type of the variable if the variable is an item. If the variable represents a table, then the declaration contains information about the types of the items in the table.

When a global variable of interest has been identified from the data dictionary display, then the RET menu from the data dictionary menubar is pulled down, and the menu item titled "Open DDE" is selected. This will cause the data dictionary entry to be displayed. When the global variable of interest has been identified from the module spec display, then the RET menu from the module spec menubar is pulled down, and the menu item titled "Open DDE" is selected. This will cause the data dictionary entry to be displayed.

A data dictionary entry may reference other data dictionary entries. This happens when the global variable represents a table or a block. In these cases, the name of the referenced data dictionary entry may be selected and the RET menu may be pulled down from the data dictionary entry menubar, and the menu item titled "Open DDE" selected. This will cause the selected data dictionary entry to be displayed in a new data dictionary entry window.

2.11 Searching Module Specs for Variable References

Global variables are associated with modules, and their module specs. A capability exists to perform a search for the modules which reference a particular global variable. This search is performed when the user selects a global variable of interest, from either the data dictionary or a module spec, pulls down the RET menu from the respective menubar, and selects the menu item titled "Display Where Ref."

After the global variable cross reference is initiated, a Unix task is invoked to perform the search within the Teamwork database. During the search, an icon is displayed with the title "REF." This icon will disappear when the search is completed, and at that time a Teamwork window will display the results of the search. The "Display Where Ref" request may be aborted using the normal Unix window procedure to quit a task represented by the REF icon. Module specs for modules identified in the cross reference display may be displayed by selecting the name of the module in the cross reference display, pulling down the RET menu from

the file menubar, and selecting the menu item titled "Show Module Spec." This will cause the respective module spec to be displayed.

Global variable cross references may also be performed across all CSCIs. This is accomplished using the "Display Where Ref All" menu item on the RET menu of the data dictionary or module spec menubar.

2.12 Printing From the RET Database

The user may obtain printouts of the process index, the data dictionary index, structure charts, module specs, data dictionary entries, any Teamwork file window that has been opened, and any expanded module source file.

2.13 Terminating Teamwork

Before terminating Teamwork, be sure that all Teamwork windows have been closed. Then, pull down the "Stop" menu from the desktop menubar and select the menu item titled "Quit". This will terminate the current Teamwork session.

3.0 Building the RET Database

3.1 CMS Rev Processor

CMS Rev consists of three (3) separate passes. These passes combine to process the CMS-2 code, to process the comments and to generate the output files used to create the RET software maintenance database. The CMS Rev processor can be executed using the shell script called build-ret which is listed in the last section (Details of Setup). This shell script takes care of deleting old versions of the build log files, and allows the user to monitor its execution with time and date stamped messages informing the user of what pass is currently being executed. The file $RET_DB_HOME/src/csci-build is used to determine which CSCIs are being processed by CMS Rev in the current execution.

3.2 CMS Rev Post-Processors

The CMS Rev post-processors augment the processing performed by CMS Rev. Two operations are performed: (1) modify some CMS Rev output files before they can be used by the RET interactive programs, and (2) analyze some CMS Rev intermediate output files to create additional output files for use by the RET interactive programs. This CMS Rev post-processing has been combined into a shell script called build-twk. This shell script should be executed once for each CSCI.

3.3 C Rev and twk_put Processors

The C Rev and twk_put processors are Teamwork programs which are used to load the Teamwork data base. C Rev uses CMS Rev output to create structure charts in the Teamwork database. twk_put uses CMS Rev output to create both module specs and data dictionary entries in the Teamwork database. A shell script called ret.script is created by CMS Rev to be used in loading the Teamwork database.

4.0 Installing the RET Processors

4.1 Teamwork Processors

The Cadre Teamwork products that must be installed include Teamwork/SD and Teamwork/C Rev. The Teamwork/C Rev Browser is not needed by the RET.

4.2 CMS Rev Processors

The CMS Rev processors consists of multiple passes as follows: cms2cdif.p3, cms2cdif.p4a, cms2cdif.p4b, and cms2cdif.p4c. These executable programs should be installed before CMS Rev can be used to build the RET database.

4.3 CMS Rev Post-Processors

The CMS Rev post-processors consists of the following programs:

build-ret
build-twk
build-csc
do-errors

5.0 Details of Set-up

5.1 Environment Variables

There is on major environment variable which must be set before running RET:

RET_DB_HOME - the directory in which all the executables and shells live

5.2 Scripts (preliminary versions)

There are several scripts which are useful in running RET (although it can be run manually). These scripts can be found in RET_DB_HOME/admin, and they are as follows:

build-ret - a multi-function script which asks for user direction upon invocation. It's current functions are: CMS Rev Pass3; CMS Rev Comment Extraction; CMS Rev Pass4; Post-Process CMS Rev; Create Teamwork Database; Dump Teamwork Database; and Restore Teamwork Database. The user is encouraged to review the script in order to get an understanding of how RET is put together.

go-ret - a script which invokes Cadre Teamwork with the proper configuration file, etc.

5.3 Directories

There are two directories which the user should set up for each model. The first is $RET_DB_HOME/src/model_name. This directory should contain the source files for the model. (This may be a soft link, if it

proves convenient.) The second directory is $RET_DB_HOME/lst/model_name. This should be created as an empty directory. CMS RET places files in there during its processing.

5.4 Files

There are two files which the user may wish to update. They are $RET_DB_HOME/dat/csci-build and $RET_DB_HOME/dat/csci-names. These are normally not needed for RET, but can be useful for rebuilding the entire system via build-ret. They should contain the model names for the system, one per line.

There is one file which the user must add to the $RET_DB_HOME/src/model_name directory: limits.txt. This file should have exactly one line in it which says "do globals".

Appendix B

Introduction to ENCORE Internal Representation

Introduction to the ENCORE Internal Representation

The Purpose of the Internal Representation.

The purpose of the ENCORE Internal Representation (or IRep, for short), is to allow the various tools in ENCORE to manipulate Ada programs in a straightforward and uniform way.

Some goals in the IRep design were:

1. There should be a logical abstract description of the IRep.
2. The IRep should be accessible via a logical interface that is independent of the physical representation of the IRep in memory.
3. There should be a clean separation between lexical information and semantic information.
4. One should be able to reconstruct the original source to an Ada program from the IRep (modulo differences in formatting).

The Logical Structure of the Internal Representation

Logically, the IRep is a tree, with some backlinks for handling references to definitions and labels, and symbol table structures to capture Ada programs. (The tree structure is quite similar to DIANA, the standard internal representation for Ada.) Each tree represents the statements in the Ada code and the symbol tables represent the scoping and visibility rules for the identifiers found in the code.

The IRep Program Tree

The tree structure consists of 'nodes' and 'attributes.' The nodes represent the information in the tree, while the attributes represent the edges of the tree.

The nodes are grouped into 'classes,' each class corresponding to a kind of Ada construct. The attributes are also grouped into named classes. As an example, consider the class of nodes corresponding to Ada assignment statements. Each such node must belong to the class 'assignment_stmt.' Furthermore, each such node must have two attributes -- one called 'target,' representing the destination of the assignment, and the other called 'source,' representing the expression to be assigned.

1. ADL: The Metalanguage Used to Describe the IRep Trees

The IRep tree is specified in a metalanguage called the Augmented Description Language, or ADL, for short.

An ADL description consists of a series of 'productions.' The productions, in turn, can be of the following kinds: stub productions, primitive productions, node productions, and class productions.

1.1 Stub Productions

Stub productions have the syntax

stub <name> ;

These productions are used to define certain special nodes that are used in processing Ada programs. There are exactly two stub productions

```
stub Empty;
stub Undefined;
```

The first production defines the 'empty' node, which is generally used to an optional attribute that is not supplied (for example, a missing 'else' clause in an 'if' statement. The second production generally indicates either an error condition or an unsuccessful operation. For example, the value returned by an unsuccessful symbol table search is the 'undefined' node.

1.2 Primitive Productions

Primitive productions have the syntax

primitive <name>;

Primitive productions are used to define external data types that are used as data at the leaves of the IRep trees. The actual productions used in the Ada IRep tree are

```
primitive Boolean;
primitive Character;
primitive Float;
primitive Integer;
primitive String;
primitive Symbol;
```

The first five productions correspond to the five predefined scalar types in the Ada package Standard. The sixth production, 'Symbol,' corresponds to the type 'Symbol,' defined in the package 'IForm_Symbols.' This Symbol type is used to represent Ada identifiers.

1.3 Node Productions

Node productions have the form

<name> => <attr1-name> : <attr1-descriptor>,
<attr2-name> : <attr2-descriptor>, ... ;

Node productions describe the internal structure of an IRep tree. The left hand side of a node production indicates the class of the node involved, while the right hand side gives the names and attributes of a node of that class.

Attributes can be either simple attributes, whose value is a node, or sequence attributes, whose value is a sequence of nodes. Simple attributes have the form

<class-name>

while sequence attributes have the form

seq of <class-name>

An example of a node production having simple attributes is given by the production for an Ada assignment statement, which is written

assignment_stmt => source : EXP,
target : REFERENCE;

This production states that each node of the class 'assignment_stmt' has two attributes, a 'source' attribute, whose value must be of class 'EXP,' and a 'target' attribute, whose value must be of class 'REFERENCE.'

As an example of sequence attributes, consider the production for an Ada else clause, which is written

else_clause => statements : seq of STMT;

This production says that each node of class 'else_clause' has the attribute 'statements,' whose value is a sequence of nodes of class STMT.

One final note. It is possible that a particular class of IRep tree node might not have any attributes. An example is the class of node that represents 'null' statements in Ada. The production for this class is written

null_stmt => ;

We use a node production, rather than a stub or primitive production because

1. there can be more than one node of class 'null_stmt' in an IRep tree structure, which rules out using a stub production, and

2. the class 'null_stmt' is not imported from another package, which rules out using a primitive production.

Class Productions

Class productions have the form

<class-name> ::= <subclass1> | <subclass2> | ... ;

The classes on the right hand side of the production are called subclasses of the class on the left hand side.

Class productions are used for two purposes:

1. To group certain classes together in a manner similar to union types in some programming languages, and
2. To enable several classes to inherit one or more attributes.

A production that satisfies the first purpose is

ACTUAL_COMPONENT ::= association | others_part | EXP;

This production says that a node of class 'ACTUAL_COMPONENT' can be either of class 'association,' class 'others_part,' or class 'EXP.' Thus the node production

aggregate => components : seq of ACTUAL_COMPONENT;

indicates that a node of class 'aggregate' has an attribute called 'components,' which can take, for its value, a sequence of nodes, each member of which must be either an 'association,' an 'others_part', or an 'EXP.'

To illustrate the second purpose of class productions, suppose we have several different kinds of nodes that possess a given attribute. In Ada, for example, package specifications, package bodies, procedure specifications, procedure bodies, etc. all have an attribute called 'designator,' which denotes the name of the unit. Rather than include a separate 'designator' attribute in each node production, we could write the two following productions:

SINGLE_DESIGNATOR_ITEM => designator : Symbol;

SINGLE_DESIGNATOR_ITEM ::= pkg_spec | pkg_bdy | proc_spec | ... ;

The attribute 'designator' will then be inherited by all subclasses of the class 'SINGLE_DESIGNATOR_ITEM.'

2. The External Representation of IRep Tree Structures

Externally, IRep tree structures are represented as one or more node structures. Node structures are represented differently, depending on the kind of node involved.

1. Stub nodes are represented by the name of the stub class; thus the two stub nodes in the Ada IRep tree are represented by

   Empty
   and
   Undefined

2. Primitive nodes are represented by the class name, followed by the primitive value, enclosed in parentheses. Some examples are

   Boolean(TRUE)
   Integer(3)
   Float(5.38)
   Character('a')
   String("Abc")
   Symbol("ABC")

3. Structure nodes may be represented by the class name, followed by the attribute names and values, enclosed within square brackets. An example is

   ```
   assignment_stmt[target n_103^,
                   source Integer(3)]
   ```

4. A structure node may be preceded by a label. This indicates that the node can appear as an attribute in more than one place in an Ada IRep tree. An example is

   ```
   n_103: named_ref[designator Symbol("x"),
                    target n_102^]
   ```

5. Finally, a labeled node can be represented simply by its label, followed by a caret, as in the reference n_102^ in the previous example. This allows us to represent circular data structures in a linear ASCII form.

3. The Ada Interface to the Internal Representation

The interface to the Ada IRep tree is provided by three packages: AdaTran_Records, Primitive_Node_Creation, and Primitive_AdaTran_Interface. The package AdaTran_Records contains the definition of IRep tree nodes and sequences; the package Primitive_Node_Creation provides functions for building IRep tree nodes; and the package
Primitive_AdaTran_Interface provides functions for accessing and changing the value of the attributes of nodes.

3.1 The Package AdaTran_Records

The package AdaTran_Records contains the following definitions.

3.1.1 The Type AdaTran_Node_Kind

The type AdaTran_Node_Kind is an enumerated type that is used to indicate the class of any given IRep tree node. The definition is

```
type AdaTran_Node_Kind is (k_UNDEFINED,
                           k_EMPTY,
                           -- Primitive Node Classes
                           k_Boolean,
                           k_Character,
                           k_Float,
                           k_Integer,
                           k_String,
                           k_Symbol,
                           -- Structured Node Classes
                           k_ABORT,
                           k_ACCEPT,
                             ...
                             ...
                           k_WITH_ELEM);
```

Note that the names of the various node kinds are all prefixed with 'k_.' This avoids any clashes with Ada reserved words. For example, ABORT and ACCEPT would class with the reserved words 'abort' and 'accept' in Ada, unless we modified them somehow.

3.1.2 The Type AdaTran_Node

The type AdaTran_Node corresponds to the IRep tree nodes for Ada. It is implemented as a pointer to a record, which contains all the attribute information for the node. Thus, we have the definition

```
type AdaTran_Node_Implementation(Kind : AdaTran_Node_Kind) is
   record
     ...
   end record;
```

and the definition

```
type AdaTran_Node is access AdaTran_Node_Implementation;
```

3.1.3 The Type Seq_Of_AdaTran_Node

The type Seq_Of_AdaTran_Node corresponds to sequences of AdaTran nodes. It is created by instantiating the generic package SEQ on the type AdaTran_Node. Thus, we have the three definitions

```
package AdaTran_Node_Seqs is new SEQ(AdaTran_Node, Eq, Equal);

subtype Seq_Of_AdaTran_Node is AdaTran_Node_Seqs.Seq;

function New_Seq_Of_AdaTran_Node return Seq_Of_AdaTran_Node
  renames AdaTran_Node_Seqs.New_Seq;
```

The generic package SEQ provides a set of routines for creating and manipulating linked lists. Instantiating this generic package for the type AdaTran_Node, makes these operations available for use on nodes. In order to use these operations on sequences of AdaTran_Node(s), it is necessary to insert the clause

```
use AdaTran_Node_Seqs;
```

in the declaration part of the unit that uses these routines.

The subtype Seq_Of_AdaTran_Node corresponds to sequences of nodes.

The function New_Seq_Of_AdaTran_Node returns the empty sequence.

3.1.4 The Functions Eq and Equal

In dealing with a complicated structures, like AdaTran_Node(s), it is sometimes necessary to make a distinction between equivalence and identity in comparing nodes. The function Eq, defined by

```
function Eq(x, y : AdaTran_Node) return Boolean;
```

returns true if and only if x and y are the same node. On the other hand, the function Equal, defined by

```
function Equal(x, y : AdaTran_Node) return Boolean;
```

returns true if and only if x and y are equivalent. In this case we require that x and y be of the same class and that all the corresponding attributes of x and y be Eq.

3.2 The Package Primitive_Node_Creation

The package Primitive_Node_Creation provides functions for constructing new nodes. These consist of the generalized node creation functions, the functions for building primitive nodes, and the functions for building structured nodes. A node, once created, can be stored in the node data base (a table in memory that holds AdaTran nodes with symbols as

the keys). The user can even specify the name under which the node should be stored. This data base is meant to provide unique names for all nodes that are attributes of two or more other nodes. Many of the node creation functions have a parameter called 'Label' or 'Node_Label,' which defaults to The_Symbol_Undefined. If the user does not specify a name, the system will generate one, if necessary.

3.2.1 The Generalized Node Creation Functions

The generalized node creation functions are

```
function Raw_AdaTran_Node(Kind : AdaTran_Node_Kind)
                    return AdaTran_Node;
```

and

```
function New_AdaTran_Node(Kind : AdaTran_Node_Kind;
                  Label : Symbol := The_Symbol_Undefined)
                   return AdaTran_Node;
```

The function Raw_AdaTran_Node simply creates a new, uninitialized instance of an AdaTran_Node. It will rarely be used by the programmer, however, since it is at a very low level and requires that the programmer devote considerable attention to low-level details.

The function New_AdaTran_Node, on the other hand, will handle many of the low level details necessary to maintain consistency in the node data base. Thus, it can be used more effectively by the programmers of ENCORE tools. The optional parameter 'Label,' indicates a name under which the node is to be stored in the node data base.

3.2.2 Functions for Building Primitive Nodes

The package Primitive_Node_Creation provides a number of functions for building primitive, or scalar, nodes, such as integers, strings, booleans, etc. Many of these functions are overloaded, in order to allow different types of parameters. Consider, for example, the function Make_Integer. There are three different versions

```
function Make_Integer(X : Integer)  return AdaTran_Node;
function Make_Integer(X : String)  return AdaTran_Node;
function Make_Integer(X : A_String) return AdaTran_Node;
```

The first Make_Integer function allows one to build a node from an actual integer. The second allows one to build a node from a string that represents an integer. Finally, the third allows one to build a node from a pointer to a string.

The other creation functions for primitive nodes are Make_Boolean, Make_Character, Make_Float, Make_String, and Make_Symbol.

There is also one function for building up sequences of symbol nodes. This is the function defined by

```
function Make_Seq_Of_Symbol(S : Seq_Of_Symbol) return Seq_Of_AdaTran_Node;
```

This function accepts a sequence of actual symbols and builds a sequence of nodes, each of type k_Symbol.

Functions for Building Structured Nodes

The functions for building structured nodes allow the user to build a complete node with all the attributes in place. Some examples are

```
function Make_Abort(p_Tasks : Seq_Of_AdaTran_Node;
                    Node_Label : Symbol := The_Symbol_Undefined)
                                          return AdaTran_Node;
   ...
   ...
function Make_Func_Spec(p_Body : AdaTran_Node;
                        p_Context : Seq_Of_AdaTran_Node;
                        p_Designator: AdaTran_Node;
                        p_Parameters : Seq_Of_AdaTran_Node;
                        p_Return_Type : AdaTran_Node;
                       Node_Label : Symbol := The_Symbol_Undefined)
                                             return AdaTran_Node;
   ...
   ..
function Make_Others(Node_Label : Symbol := The_Symbol_Undefined)
                                            return AdaTran_Node;.
```

The names of parameters that correspond to attribute values are all prefixed with 'p_.' This avoids any clashes with Ada reserved words. For example, several classes of node contain an attribute called 'type.' This would cause a conflict with the reserved word 'type' in Ada, unless we altered the name somehow.

3.3 The Package Primitive_AdaTran_Interface

The package Primitive_AdaTran_Interface provides routines for accessing and manipulating AdaTran_Nodes.

3.3.1 The Function Kind

The function Kind, defined by

```
function Kind(x : AdaTran_Node) return AdaTran_Node;
```

allows the user to query a node as to its class. Quite often the various ENCORE tools will use a case-statement based on the result of Kind(x), then perform different operations depending on the actual kind of the node.

3.3.2 Accessing Primitive Nodes

Primitive nodes can't be altered, so the only operation available is to retrieve the actual primitive values from the nodes. For example, we can retrieve the integer value of an integer node. The actual functions are

```
function As_Boolean(N : AdaTran_Node) return Boolean;
function As_Character(N : AdaTran_Node) return Character;
function As_Float(N : AdaTran_Node) return Float;
function As_Integer(N : AdaTran_Node) return Integer;
function As_String(N : AdaTran_Node) return String;
function As_A_String(N : AdaTran_Node) return A_String;
function As_Symbol(N : AdaTran_Node) return Symbol;
```

The function As_A_String needs some additional comments. The type A_String is an access type whose values are pointers to strings (A_String is described in the package Basic_Strings). With string nodes, it is important to be able to view the string value of a string node as either an actual string or as a pointer to a string. This is because the type String in Ada is an unconstrained array type, which is inconvenient to use in some contexts.

Finally, there is a function defined by

```
function As_Seq_Of_Symbol(S : Seq_Of_AdaTran_Node)
                                    return Seq_Of_AdaTran_Node;
```

This function takes a sequence of AdaTran_Node(s), all presumed to be of type k_Symbol, and returns a sequence of Symbols. As such, it is the reverse of the function Make_Seq_Of_Symbol, defined in the package Primitive_Node_Creation.

3.3.3 Accessing Structure Nodes

For each attribute name, there are two corresponding functions, a 'Get_' function and a 'Set_' function. The Get function retrieves the attribute of the given name, while the Set function assigns a value to the attribute. Two examples are

```
Get_Type(N : AdaTran_Node) return AdaTran_Node;
Set_Type(N : AdaTran_Node; To_Be : AdaTran_Node);
```

and

```
Get_Declarations(N : AdaTran_Node) return Seq_Of_AdaTran_Node;
Set_Declarations(N : AdaTran_Node; To_Be : Seq_Of_AdaTran_Node);
```

The first two functions provide access to the 'type' attribute of any typed node, such as a var_decl, const_decl, etc.

The last two functions provide access to the 'declarations' attribute for any node corresponding to a scope. These include nodes of class pkg_spec, pkg_bdy, block, etc.

Appendix C

ADL Description of the Ada Internal Representation

-- ADL Description of the Ada Internal Representation

```
module AdaTran is

 -- Primitive Node Types

 primitive Boolean;
 primitive Character;
 primitive Float;
 primitive Integer;
 primitive String;
 primitive Symbol;

 stub Empty;
 stub Undefined;

-- Structured Classes

-- 2.8 pragmas

 pragma =>
  designator : Symbol,
  parameters : seq of EXP_OR_ASSOCIATION;

 EXP_OR_ASSOCIATION ::= EXP | association;

-- 3. declarations and types

-- 3.1 declarations

 DECL ::= pragma | use_elem |
          MULTIPLE_DESIGNATORS_ITEM | REP | SINGLE_DESIGNATOR_ITEM;

-- 3.2 objects and named numbers

 MULTIPLE_DESIGNATORS_ITEM ::= num_decl | var_decl | const_decl;

 EXP_OR_EMPTY ::= EXP | Empty;

 SUBTYPE_INDICATION := constrained_reference REFERENCE;

 num_decl =>
  designators   : seq of Symbol,
  initial_value : EXP_OR_EMPTY,
  referencers   : seq of named_ref,
```

```
  type        : SUBTYPE_INDICATION;

 var_decl =>
  constraints  : seq of CONSTRAINT,
  designators  : seq of Symbol,
  initial_value : EXP_OR_EMPTY,
  referencers  : seq of named_ref,
  type        : SUBTYPE_INDICATION;

 const_decl =>
  constraints  : seq of CONSTRAINT,
  designators  : seq of Symbol,
  initial_value : EXP_OR_EMPTY,
  referencers  : seq of named_ref,
  type        : SUBTYPE_INDICATION;

-- 3.3 types and subtypes

-- 3.3.1 type declarations

 SINGLE_DESIGNATOR_ITEM ::= type_decl | subtype_decl;

 type_decl =>
  designator : Symbol,
  info      : TYPE_INFO,
  referencer : direct_ref;


-- 3.3.2 subtype declarations

 subtype_decl =>
  base_type   : SUBTYPE_INDICATION,
  constraints : seq of CONSTRAINT,
  designator  : Symbol,
  referencer  : direct_ref;


-- 3.4 derived type defintions

 TYPE_INFO       ::= derived_type_info;

 derived_type_info =>
  base_type   : SUBTYPE_INDICATION,
  constraints : seq of CONSTRAINT;
```

```
-- 3.5 scalar types

 TYPE_INFO ::= enumerated_type_info;

 enumerated_type_info =>
  values : seq of enumeration_literal;

 enumeration_literal =>
  base_type : SUBTYPE_INDICATION,
  type      : direct_ref,
  value     : SYMBOL_OR_CHARACTER;

 SYMBOL_OR_CHARACTER ::= Symbol | Character;

-- 3.5.4 integer types

 TYPE_INFO ::= integer_type_info;

 integer_type_info =>
  range : SIMPLE_RANGE;

-- 3.5.9 real types

 TYPE_INFO ::= REAL_TYPE_INFO;

 REAL_TYPE_INFO ::= float_type_info;
 float_type_info =>
  digits : EXP,
  range  : SIMPLE_RANGE_OR_EMPTY;

 REAL_TYPE_INFO ::= fixed_type_info;
 fixed_type_info =>
  delta : EXP,
  range : SIMPLE_RANGE_OR_EMPTY;

 SIMPLE_RANGE_OR_EMPTY ::= SIMPLE_RANGE | Empty;

-- 3.6 array types

 TYPE_INFO     ::= array_type_info;

 array_type_info =>
  base_type : SUBTYPE_INDICATION,
  ranges    : seq of RANGE;

 RANGE ::= discrete_range;
```

```
SIMPLE_RANGE ::= discrete_range;

discrete_range =>
  base_type : SUBTYPE_INDICATION,
  max       : EXP,
  min       : EXP;

RANGE ::= index_constraint;

index_constraint =>
  base_type : SUBTYPE_INDICATION,
  max       : EXP,
  min       : EXP;

RANGE ::= universal_index_range;

universal_index_range =>
  base_type : SUBTYPE_INDICATION,
  max       : EXP,
  min       : EXP;

RANGE ::= universal_integer_range;

universal_integer_range =>
  base_type : SUBTYPE_INDICATION,
  max       : EXP,
  min       : EXP;

RANGE ::= REFERENCE;

-- 3.7 record types

TYPE_INFO ::= record_type_info;

record_type_info =>
  components   : seq of component_decl,
  discriminant : seq of component_decl;

MULTIPLE_DESIGNATORS_ITEM ::= component_decl;
COMPONENT    ::= component_decl | pragma;

component_decl =>
  constraints   : seq of CONSTRAINT,
  designators   : seq of Symbol,
  initial_value : EXP_OR_EMPTY,
  referencers   : seq of named_ref,
  type          : SUBTYPE_INDICATION;
```

```
COMPONENT ::= null_component;

null_component => ;

COMPONENT ::= variant_part;
variant_part =>
  discriminator : named_ref,
  variants      : seq of variant;

variant =>
  choices    : seq of CHOICE_OR_OTHERS,
  components : seq of COMPONENT;

CHOICE_OR_OTHERS    ::= EXP | GENERAL_DISCRETE_RANGE | others;

others =>;
```

-- 3.8 access types

```
TYPE_INFO ::= pointer_type_info;

pointer_type_info =>
  base_type : SUBTYPE_INDICATION;
```

-- 3.8.1 Incomplete Type Declarations

```
TYPE_INFO ::= TYPE_STUB;

TYPE_STUB ::= incomplete_type_info;

incomplete_type_info =>
  completion   : DIRECT_REF_OR_EMPTY,
  discriminant : seq of component_decl;

TYPE_STUB ::= private_type_info;

private_type_info =>
  completion   : DIRECT_REF_OR_EMPTY,
  discriminant : seq of component_decl;

TYPE_STUB ::= limited_private_type_info;

limited_private_type_info =>
  completion   : DIRECT_REF_OR_EMPTY,
  discriminant : seq of component_decl;
```

```
TYPE_INFO ::= type_completion_info;
type_completion_info =>
  info : TYPE_INFO,
  stub : DIRECT_REF_OR_EMPTY;

DIRECT_REF_OR_EMPTY ::= direct_ref | Empty;
```

-- 3.9 declarative parts

-- 4 names and expressions

-- 4.1 names

-- 4.1.1 indexed components

```
REFERENCE ::= indexed_ref;
indexed_ref =>
  indices         : seq of EXP,
  representations : seq of REP,
  target          : EXP;
```

-- 4.1.2 slices

```
slice =>
  range  : GENERAL_DISCRETE_RANGE,
  target : EXP;

GENERAL_DISCRETE_RANGE ::= constrained_reference | REFERENCE | SIM-
PLE_RANGE;
```

-- 4.1.3 selected components

```
REFERENCE ::= component_ref;

component_ref =>
  component       : EXP,
  representations : seq of REP,
  target          : EXP;
```

-- 4.1.4 attributes

```
SIMPLE_RANGE ::= attribute;

attribute =>
```

```
  designator : Symbol,
  exp        : EXP;

 SIMPLE_RANGE ::= attribute_call;

 attribute_call =>
  attribute : attribute,
  exp       : EXP;

-- 4.2 literals

 EXP ::= LITERAL;

 LITERAL := Boolean | Integer | Float | Symbol | Character | String;

-- 4.3 aggregates

 LITERAL ::= aggregate;
 aggregate =>
  components : seq of ACTUAL_COMPONENT;

 ACTUAL_COMPONENT ::= association | others_part | EXP;

 others_part =>
  exp : EXP;

-- 4.4 expressions

 EXP ::= REFERENCE;

-- 4.4.B relations

 EXP ::= membership;

 membership =>
  exp · EXP,
  op  : MEMBERSHIP_OP,
  set : discrete_range;

 MEMBERSHIP_OP ::= in_op | not_in;

 in_op => ;
 not_in => ;

-- 4.5 operators and expression evaluation
```

-- See Function Calls

-- 4.6 type conversions

```
EXP ::= QUAL_CONV;
QUAL_CONV ::= conversion;

conversion =>
  exp  : EXP,
  type : SUBTYPE_INDICATION;
```

-- 4.7 qualified expressions

```
QUAL_CONV ::= qualified_expression;

qualified_expression =>
  exp  : EXP,
  type : SUBTYPE_INDICATION;
```

-- 4.8 allocators

```
EXP ::= ALLOCATOR;
ALLOCATOR ::= uninitialized_allocator;

uninitialized_allocator =>
  constraints : seq of CONSTRAINT,
  object_type : SUBTYPE_INDICATION,
  type        : SUBTYPE_INDICATION;

ALLOCATOR ::= initialized_allocator;

initialized_allocator =>
  constraints : seq of CONSTRAINT,
  expr        : qualified_expression,
  type        : SUBTYPE_INDICATION;

EXP ::= null_exp;

null_exp => ;
```

-- 5 Statements

```
STMT ::= pragma;
STMT ::= labeled_stmt;
```

```
labeled_stmt =>
  labels     : seq of Symbol,
  referencers : seq of named_ref,
  statement   : STMT;

STMT ::= null_stmt;

null_stmt => ;
```

-- 5.2 assignment statement

```
STMT ::= assignment_stmt;

assignment_stmt =>
  target : REFERENCE,
  source : EXP;
```

-- 5.3 if statements

```
STMT ::= if_stmt;
if_stmt =>
  then_part  : then_clause,
  else_parts : ELSES_OR_EMPTY;

then_clause =>
  cond      : EXP,
  statements : seq of STMT;

ELSES_OR_EMPTY ::= elses_part | Empty;

elses_part =>
  else_part : ELSE_CLAUSE_OR_EMPTY,
  elsifs    : seq of elsif_clause;

elsif_clause =>
  cond      : EXP,
  statements : seq of STMT;

ELSE_CLAUSE_OR_EMPTY ::= else_clause | Empty;

else_clause =>
statements : seq of STMT;
```

-- 5.4 case statements

```
STMT ::= case_stmt;
```

```
case_stmt =>
  alternatives : seq of altern,
  case_exp     : EXP;

altern =>
  choices    : seq of CHOICE_OR_OTHERS,
  statements : seq of STMT;
```

-- 5.5 loop statements

```
STMT              ::= loop_stmt;

loop_stmt =>
  iterator   : ITERATOR,
  label      : Symbol,
  referencer : direct_ref,
  statements : seq of STMT;

ITERATOR ::= while_iter;

while_iter =>
  condition : EXP;

ITERATOR ::= for_iter;

for_iter =>
  init_and_end : GENERAL_DISCRETE_RANGE,
  referencers  : seq of named_ref,
  variable     : Symbol;

ITERATOR ::= reverse_iter;

reverse_iter =>
  init_and_end : GENERAL_DISCRETE_RANGE,
  referencers  : seq of named_ref,
  variable     : Symbol;
```

-- 5.6 block statements

```
STMT ::= block;

block =>
  declarations      : seq of DECL,
  exception_handler : seq of altern,
  label             : Symbol,
```

```
  referencer      : direct_ref,
  statements      : seq of STMT;

-- 5.7 exit_statements

 STMT ::= exit_stmt;

 exit_stmt =>
  level          : REFERENCE_OR_EMPTY,
  when_condition : EXP_OR_EMPTY;

 REFERENCE_OR_EMPTY ::= REFERENCE | Empty;

-- 5.8 return statements

 STMT ::= return_stmt;
 return_stmt =>
  value : EXP_OR_EMPTY;

-- 5.9 goto statements

 STMT ::= goto_stmt;
 goto_stmt =>
  target : REFERENCE;

-- 6 subprograms

-- 6.1 subprogram declarations

 SINGLE_DESIGNATOR_ITEM ::= func_spec;

 func_spec =>
  body       : DIRECT_REF_OR_EMPTY,
  context    : seq of CONTEXT_ELEM,
  designator : Symbol,
  parameters : seq of FORMAL,
  referencer : direct_ref,
  return_type : direct_ref;

 SINGLE_DESIGNATOR_ITEM ::= proc_spec;

 proc_spec =>
  body       : DIRECT_REF_OR_EMPTY,
  context    : seq of CONTEXT_ELEM,
  designator : Symbol,
  parameters : seq of FORMAL,
```

```
  referencer : direct_ref;

-- 6.1.C formal part

 FORMAL ::= in_formal | out_formal | inout_formal;

 in_formal =>
  designators   : seq of Symbol,
  initial_value : EXP_OR_EMPTY,
  referencers   : seq of named_ref,
  type          : SUBTYPE_INDICATION;

 FORMAL ::= out_formal;
 out_formal =>
  designators : seq of Symbol,
  referencers : seq of named_ref,
  type        : SUBTYPE_INDICATION;

 FORMAL           ::= inout_formal;
 GENERIC_PARAMETER ::= inout_formal;
 inout_formal =>
  designators : seq of Symbol,
  referencers : seq of named_ref,
  type        : SUBTYPE_INDICATION;

-- 6.3 subprogram bodies

 SINGLE_DESIGNATOR_ITEM ::= func_bdy;

 func_bdy =>
  context         : seq of CONTEXT_ELEM,
  declarations    : seq of DECL,
  designator      : Symbol,
  exception_handler : seq of altern,
  parameters      : seq of FORMAL,
  referencer      : direct_ref,
  return_type     : direct_ref,
  spec            : DIRECT_REF_OR_EMPTY,
  statements      : seq of STMT;

 SINGLE_DESIGNATOR_ITEM ::= proc_bdy;

 proc_bdy =>
  context         : seq of CONTEXT_ELEM,
  declarations    : seq of DECL,
  designator      : Symbol,
```

```
  exception_handler : seq of altern,
  parameters        : seq of FORMAL,
  referencer        : direct_ref,
  spec              : DIRECT_REF_OR_EMPTY,
  statements        : seq of STMT;

-- 6.4 subprogram calls

 STMT ::= proc_call;

 proc_call =>
  parameters : seq of EXP_OR_ASSOCIATION,
  proc       : REFERENCE;

 EXP ::= function_call;

 function_call =>
  function   : REFERENCE,
  parameters : seq of EXP_OR_ASSOCIATION;


-- 7 packages

-- 7.1 package structure

 SINGLE_DESIGNATOR_ITEM ::= pkg_spec;

 pkg_spec =>
  body               : DIRECT_REF_OR_EMPTY,
  context            : seq of CONTEXT_ELEM,
  declarations       : seq of DECL,
  designator         : Symbol,
  private_declarations : seq of DECL,
  referencer         : direct_ref;

 SINGLE_DESIGNATOR_ITEM ::= pkg_bdy;

 pkg_bdy =>
  context          : seq of CONTEXT_ELEM,
  declarations     : seq of DECL,
  designator       : Symbol,
  exception_handler : seq of altern,
  referencer       : direct_ref,
  spec             : DIRECT_REF_OR_EMPTY,
  statements       : seq of STMT;
```

-- 7.4 private type and deferred constant declarations

```
MULTIPLE_DESIGNATORS_ITEM ::= deferred_const_decl;

deferred_const_decl =>
  decl       : const_decl,
  designators : seq of Symbol,
  referencers : seq of named_ref,
  type       : SUBTYPE_INDICATION;
```

-- 8 visibility rules

-- 8.4 use clauses

```
DECL         ::= use_elem;
CONTEXT_ELEM ::= use_elem;

use_elem =>
  items : seq of Symbol;
```

-- 8.5 renaming declarations

```
MULTIPLE_DESIGNATORS_ITEM ::= exception_rename;

exception_rename =>
  designators : seq of Symbol,
  item       : REFERENCE;

SINGLE_DESIGNATOR_ITEM ::= func_rename;

func_rename =>
  designator  : Symbol,
  item       : REFERENCE,
  parameters  : seq of FORMAL,
  referencer  : direct_ref,
  return_type : direct_ref;

MULTIPLE_DESIGNATORS_ITEM ::= object_rename;

object_rename =>
  designators : seq of Symbol,
item       : REFERENCE;

SINGLE_DESIGNATOR_ITEM ::= pkg_rename;

pkg_rename =>
```

```
  designator : Symbol,
  item       : REFERENCE,
  referencer : direct_ref;

SINGLE_DESIGNATOR_ITEM ::= proc_rename;

proc_rename =>
  designator : Symbol,
  item       : REFERENCE,
  parameters : seq of FORMAL,
  referencer : direct_ref;

-- 9 tasks

-- 9.1 task specifications and task bodies

SINGLE_DESIGNATOR_ITEM ::= task_type_decl;

task_type_decl =>
  designator : Symbol,
  referencer : direct_ref,
  spec       : DIRECT_REF_OR_EMPTY;

SINGLE_DESIGNATOR_ITEM ::= task_spec;
TYPE_INFO              ::= task_spec;
task_spec =>
  body         : DIRECT_REF_OR_EMPTY,
  context      : seq of CONTEXT_ELEM,
  declarations : seq of DECL,
  designator   : Symbol,
  referencer   : direct_ref;


SINGLE_DESIGNATOR_ITEM ::= task_bdy;

task_bdy =>
  context         : seq of CONTEXT_ELEM,
  declarations    : seq of DECL,
  designator      : Symbol,
  exception_handler : seq of altern,
  referencer      : direct_ref,
  spec            : DIRECT_REF_OR_EMPTY,
  statements      : seq of STMT;

-- 9.5 entries, entry calls and accept statements
```

```
SINGLE_DESIGNATOR_ITEM ::= entry;

entry =>
  designator : Symbol,
  parameters : seq of FORMAL,
  range      : RANGE_OR_EMPTY,
  referencer : direct_ref;

RANGE_OR_EMPTY ::= RANGE | Empty;

entry_call =>
  entry      : direct_ref,
  index      : EXP_OR_EMPTY,
  parameters : seq of EXP_OR_ASSOCIATION;

STMT ::= accept;
accept =>
  entry      : REFERENCE,
  index      : EXP_OR_EMPTY,
  parameters : seq of FORMAL,
  referencer : direct_ref,
  statements : seq of STMT;

-- 9.6 delay statements, duration and time

delay =>
  exp : EXP;

-- 9.7 select statements

-- 9.7.1 selective waits

select =>
  select_clauses : seq of SELECT_CLAUSE_ELEM,
  statements     : seq of STMT;

SELECT_CLAUSE_ELEM ::= pragma | select_clause;

select_clause =>
  cond       : EXP,
  statements : seq of STMT;

STMT ::= terminate;

terminate => ;
```

-- 9.7.2 conditional entry calls

STMT ::= ENTRY_STMT;

ENTRY_STMT ::= cond_entry;

cond_entry =>
  failure_statements : seq of STMT,
  success_statements : seq of STMT;

-- 9.7.3 timed entry calls

ENTRY_STMT ::= timed_entry;

timed_entry =>
  failure_statements : seq of STMT,
  success_statements : seq of STMT;

-- 9.10 abort statements

STMT ::= abort;

abort =>
  tasks : seq of REFERENCE;

-- 10 program structure and compilation issues

-- 10.1 compilation units - library units

UNIT_DECL ::= GENERIC_INSTANTIATION ACTUAL_SPEC ACTUAL_BODY;

ACTUAL_SPEC ::=
  func_spec |
  func_instantiation |
  generic_func_spec |
  generic_pkg_spec |
  generic_proc_spec |
  pkg_spec |
  pkg_instantiation |
  pkg_spec |
  proc_spec |
  proc_instantiation |
  task_spec;

ACTUAL_BODY ::=
  func_bdy |

```
  func_instantiation_bdy |
  generic_func_bdy |
  generic_pkg_bdy |
  generic_proc_bdy |
  pkg_bdy |
  pkg_instantiation_bdy |
  proc_bdy |
  proc_instantiation_bdy |
  task_bdy;

 CONTEXT_ELEM ::= with_elem;
--CONTEXT_ELEM ::= use_elem;

 with_elem =>
  items : seq of Symbol;

-- 10.2 subunits of compilation units

 SINGLE_DESIGNATOR_ITEM ::= stub;

 stub =>
  designator : Symbol,
  referencer : direct_ref,
  spec       : DIRECT_REF_OR_EMPTY,
  subunit    : DIRECT_REF_OR_EMPTY;

 SINGLE_DESIGNATOR_ITEM ::= subunit;

 subunit =>
  body       : DIRECT_REF_OR_EMPTY,
  designator : Symbol,
  referencer : direct_ref,
  spec       : DIRECT_REF_OR_EMPTY,
  stub       : DIRECT_REF_OR_EMPTY;

-- 11 exceptions

-- 11.1 exception declarations

 MULTIPLE_DESIGNATORS_ITEM ::= exception_decl;

 exception_decl =>
  designators : seq of Symbol,
  referencers : seq of named_ref;
```

```
-- 11.2 exception handlers


-- 11.3 raise statements

 STMT ::= raise_stmt;

 raise_stmt =>
  exception : named_ref;

-- 12 generic program units

 GENERIC_UNIT := generic_func_spec | generic_proc_spec |
                 generic_func_bdy | generic_proc_bdy |
                 generic_pkg_spec;

 GENERIC_PARAMETER ::= generic_type_param |
                       in_formal |
                       inout_formal |
                      GENERIC_SUBPROGRAM_PARAM;

 SINGLE_DESIGNATOR_ITEM ::= generic_type_param;

 generic_type_param =>
  designator : Symbol,
  info       : GENERIC_TYPE_INFO,
  referencer : direct_ref;

 GENERIC_TYPE_INFO ::= generic_discrete_info | generic_integer_info |
                       generic_float_info   | generic_fixed_info  |
                       TYPE_INFO;

 generic_discrete_info => ;
 generic_integer_info => ;
 generic_float_info => ;
 generic_fixed_info => ;

 GENERIC_SUBPROGRAM_PARAM ::= generic_func_param | generic_proc_param;

 generic_func_param =>
  default_subprogram : SYMBOL_BOX_SUBPROGRAM_OR_EMPTY,
  designator         : Symbol,
 parameters          : seq of FORMAL,
  referencer         : direct_ref,
  return_type        : direct_ref;
```

```
generic_proc_param =>
  default_subprogram : SYMBOL_BOX_SUBPROGRAM_OR_EMPTY,
  designator       : Symbol,
  parameters       : seq of FORMAL,
  referencer       : direct_ref;

 SYMBOL_BOX_SUBPROGRAM_OR_EMPTY ::= box_subprogram | Empty | Sym-
bol;

 box_subprogram => ;

 SINGLE_DESIGNATOR_ITEM ::= generic_func_spec;

 generic_func_spec =>
  body       : DIRECT_REF_OR_EMPTY,
  context    : seq of CONTEXT_ELEM,
  designator : Symbol,
  g_parameters : seq of GENERIC_PARAMETER,
  parameters : seq of FORMAL,
  referencer : direct_ref,
  return_type : direct_ref;

 SINGLE_DESIGNATOR_ITEM ::= generic_proc_spec;

 generic_proc_spec =>
  body       : DIRECT_REF_OR_EMPTY,
  context    : seq of CONTEXT_ELEM,
  designator : Symbol,
  g_parameters : seq of GENERIC_PARAMETER,
  parameters : seq of FORMAL,
  referencer : direct_ref;

 SINGLE_DESIGNATOR_ITEM   ::= generic_pkg_spec;

 generic_pkg_spec =>
  body             : DIRECT_REF_OR_EMPTY,
  context          : seq of CONTEXT_ELEM,
  declarations     : seq of DECL,
  designator       : Symbol,
  g_parameters     : seq of GENERIC_PARAMETER,
  private_declarations : seq of DECL,
  referencer       : direct_ref;


 SINGLE_DESIGNATOR_ITEM ::= generic_func_bdy;

 generic_func_bdy =>
```

```
  context         : seq of CONTEXT_ELEM,
  declarations    : seq of DECL,
  designator      : Symbol,
  exception_handler : seq of altern,
  g_parameters    : seq of GENERIC_PARAMETER,
  parameters      : seq of FORMAL,
  referencer      : direct_ref,
  return_type     : direct_ref,
  spec            : DIRECT_REF_OR_EMPTY,
  statements      : seq of STMT;

SINGLE_DESIGNATOR_ITEM ::= generic_proc_bdy;

generic_proc_bdy =>
  context         : seq of CONTEXT_ELEM,
  declarations    : seq of DECL,
  designator      : Symbol,
  exception_handler : seq of altern,
  g_parameters    : seq of GENERIC_PARAMETER,
  parameters      : seq of FORMAL,
  referencer      : direct_ref,
  spec            : DIRECT_REF_OR_EMPTY,
  statements      : seq of STMT;

SINGLE_DESIGNATOR_ITEM ::= generic_pkg_bdy;

generic_pkg_bdy =>
  context         : seq of CONTEXT_ELEM,
  declarations    : seq of DECL,
  designator      : Symbol,
  exception_handler : seq of altern,
  referencer      : direct_ref,
  spec            : DIRECT_REF_OR_EMPTY,
  statements      : seq of STMT;
```

-- 12.3 generic instantiation

```
GENERIC_ACTUAL_PARAMETER := EXP; -- for now

SINGLE_DESIGNATOR_ITEM ::= func_instantiation;
GENERIC_INSTANTIATION  ::= func_instantiation;

func_instantiation =>
  body      : DIRECT_REF_OR_EMPTY,
  context   : seq of CONTEXT_ELEM,
  designator : Symbol,
```

```
 g_actuals   : seq of GENERIC_ACTUAL_PARAMETER,
 instance_of : REFERENCE,
 parameters  : seq of FORMAL,
 referencer  : direct_ref,
 return_type : direct_ref,
 spec        : DIRECT_REF_OR_EMPTY;

SINGLE_DESIGNATOR_ITEM ::= proc_instantiation;
GENERIC_INSTANTIATION  ::= proc_instantiation;

proc_instantiation =>
 body        : DIRECT_REF_OR_EMPTY,
 context     : seq of CONTEXT_ELEM,
 designator  : Symbol,
 g_actuals   : seq of GENERIC_ACTUAL_PARAMETER,
 instance_of : REFERENCE,
 parameters  : seq of FORMAL,
 referencer  : direct_ref,
 spec        : DIRECT_REF_OR_EMPTY;

SINGLE_DESIGNATOR_ITEM ::= pkg_instantiation;
GENERIC_INSTANTIATION  ::= pkg_instantiation;

pkg_instantiation =>
 body        : DIRECT_REF_OR_EMPTY,
 context     : seq of CONTEXT_ELEM,
 designator  : Symbol,
 g_actuals   : seq of GENERIC_ACTUAL_PARAMETER,
 instance_of : REFERENCE,
 referencer  : direct_ref,
 spec        : DIRECT_REF_OR_EMPTY;

SINGLE_DESIGNATOR_ITEM ::= func_instantiation_bdy;
GENERIC_INSTANTIATION  ::= func_instantiation_bdy;
func_instantiation_bdy =>
 context         : seq of CONTEXT_ELEM,
 declarations    : seq of DECL,
 designator      : Symbol,
 exception_handler : seq of altern,
 g_actuals       : seq of GENERIC_ACTUAL_PARAMETER,
 instance_of     : REFERENCE,
 referencer      : direct_ref,
 spec            : DIRECT_REF_OR_EMPTY,
 statements      : seq of STMT;

SINGLE_DESIGNATOR_ITEM ::= proc_instantiation_bdy;
GENERIC_INSTANTIATION  ::= proc_instantiation_bdy;
```

```
proc_instantiation_bdy =>
  context         : seq of CONTEXT_ELEM,
  declarations    : seq of DECL,
  designator      : Symbol,
  exception_handler : seq of altern,
  g_actuals       : seq of GENERIC_ACTUAL_PARAMETER,
  instance_of     : REFERENCE,
  referencer      : direct_ref,
  spec            : DIRECT_REF_OR_EMPTY,
  statements      : seq of STMT;

SINGLE_DESIGNATOR_ITEM ::= pkg_instantiation_bdy;
GENERIC_INSTANTIATION  ::= pkg_instantiation_bdy;

pkg_instantiation_bdy =>
  context         : seq of CONTEXT_ELEM,
  declarations    : seq of DECL,
  designator      : Symbol,
  exception_handler : seq of altern,
  g_actuals       : seq of GENERIC_ACTUAL_PARAMETER,
  instance_of     : REFERENCE,
  referencer      : direct_ref,
  spec            : DIRECT_REF_OR_EMPTY,
  statements      : seq of STMT;
```

-- 13 representation clauses and implementation dependent features

-- 13.1 representation clauses

```
REP ::= record_rep | EXP_REP;

type_attribute =>
  designator     : Symbol,
  representations : seq of REP,
  type           : direct_ref;

(ADL EXP_REP => exp EXP)

(ADL EXP_REP := address_rep length_clause enumeration_rep)-- 13.3
```

-- 13.2 length clauses

```
EXP_REP ::= length_clause;
length_clause =>
  exp   : EXP, -- exp is a simple expression
```

```
  target : REFERENCE_OR_TYPE_ATTRIBUTE;

-- 13.3 enumeration representation clauses

 EXP_REP ::= enumeration_rep;
 enumeration_rep =>
  exp    : EXP,  -- exp is an aggregate
  target : REFERENCE_OR_TYPE_ATTRIBUTE;

-- 13.4 record representation clauses

 ALIGNMENT_OR_EMPTY ::= alignment | Empty;
 alignment =>
  at_mod  : EXP_OR_EMPTY,
  pragmas : seq of pragma;

 REP ::= record_rep;
 record_rep =>
  alignment     : ALIGNMENT_OR_EMPTY,
  component_reps : seq of COMPONENT_REP_ELEMENT,
  target        : REFERENCE_OR_TYPE_ATTRIBUTE;

 COMPONENT_REP_ELEMENT ::= component_rep | pragma;

 component_rep =>
  at_exp    : EXP,
  designator : Symbol,
  range     : RANGE;

-- 13.5 address clauses

 EXP_REP ::= address_rep;
 address_rep =>
  exp    : EXP,
  target : REFERENCE_OR_TYPE_ATTRIBUTE;

 REFERENCE_OR_TYPE_ATTRIBUTE ::= REFERENCE | type_attribute;

 type_attribute =>
  designator     : Symbol,
  representations : seq of REP,
  type           : direct_ref;

-- 13.8 machine code insertions

 machine_code =>
```

```
  exp  : EXP,
  type : REFERENCE;

-- 14 input-output

-- X.1 Constraints

 CONSTRAINT ::= RANGE_CONSTRAINT | array_constraint | Empty;
 RANGE_CONSTRAINT ::= EXP | RANGE;

 array_constraint =>
  element_constraints : seq of CONSTRAINT,
  range_constraints   : seq of RANGE;

-- X.1 References

 REFERENCE ::= direct_ref   named_ref  indexed_ref
          component_ref object_ref pointer_deref)

 REFERENCE ::= direct_ref;

 direct_ref =>
  representations : seq of REP,
  target          : DECL_OR_STMT; -- points to single decls

 DECL_OR_STMT ::= DECL | STMT;

 REFERENCE ::= object_ref;

 object_ref =>
  representations : seq of REP,
  target          : EXP;

 REFERENCE        ::= named_ref;

 named_ref =>
  designator     : Symbol,
  representations : seq of REP,
  target          : REFERENCE;

 REFERENCE ::= pointer_deref;

 pointer_deref =>  -- the ".all" construct
  representations : seq of REP,
  target          : REFERENCE;
```

```
 EXP_OR_ASSOCIATION ::= association;
 ACTUAL_COMPONENT   ::= association;
 association =>
  names : seq of EXP,
  value : EXP;

 GENERAL_DISCRETE_RANGE ::= constrained_reference;
 SUBTYPE_INDICATION     ::= constrained_reference;

 constrained_reference =>
  constraint : CONSTRAINT,
  target     : REFERENCE;

 unconstrained =>
  base_type : SUBTYPE_INDICATION;

end module;
```

Appendix D

Introduction to ENCORE Symbol Table

Introduction to the ENCORE Symbol Table

1. The Purpose of the ENCORE Symbol Table

The purpose of the ENCORE Symbol Table is to allow the various tools in ENCORE to access definitions in Ada programs within the context of particular scopes.

Some goals in the symbol table design were:

1. The symbol table should be incrementally updatable. One should be able to add, remove, or replace symbol table entries at any time, not just when the program is being read in initially.
2. From any point in any scope of the program, the symbol table should appear logically the same as if one were processing the Ada code at that point in the program.
3. Any tool that works on the IRep should be able to access the symbol table.
4. More than one tool should be able to access the symbol table simultaneously, even within multiple scopes.

1.1 Basic Definitions

Symbol tables are used to store information that can be referenced from more than one point in a program. Each item of information in a symbol table consists of two parts, a 'key,' which gives a name to the item, and a 'value,' which gives the actual information. Adding the item with key 'k' and value 'v' to a symbol table is called storing the value 'v' under the key 'k.'

Some of symbol table terminology has been used with slightly different meanings in the current literature. This report will adopt the following meanings for the common symbol table terms.

1. The phrase 'the symbol table' means the entire symbol table structure associated with a particular Ada program.
2. The non-specific term 'symbol table' will denote a table of key/value pairs. The keys will correspond to identifiers or characters in Ada, while the values will correspond to definitions in Ada. For flexibility in modifying Ada programs, the values will be references (nodes of type k_DIRECT_REF or k_NAMED_REF) rather than actual definitions.
3. The term 'scope' will mean a symbol table associated with a segment of a particular Ada program unit. Thus, a scope could hold information about a specification, a private part, or a body.
4. A 'search' is an object that is used for accessing symbol tables. A search object includes all the local and nesting information necessary for searching through the symbol table for an Ada program.
5. The 'base table' of a search object indicates the scope in which searching will start.

6. Since there can be several entries with the same key in the symbol table for an Ada program, it is necessary to keep track of which entries have been found and which are left to search. The 'search cursor' keeps track of this information.
7. One is said to be 'in' a particular scope if the base table of the search object corresponds to that particular scope.

1.2 A Simplified View of the Symbol Table

The various symbol table packages offer the programmer a great deal of power and flexibility; however, most programmers of ENCORE tools will only be interested in a relatively small set of operations. These include:

- determining the current scope,
- creating a new scope within a given scope,
- entering a previously created scope,
- leaving a given scope (returning to the parent scope),
- entering the scope associated with a given declaration,
- retrieving the local entry (or entries) associated with a given key in a given scope, and
- retrieving the entry (or entries) associated with a given key, visible in the given scope (i.e., those entries that are either in the local scope, in any of the parent scopes, or made visible via *'with'* or *'use' clauses*).
- adding an entry to a given scope,
- removing an entry from a given scope,
- replacing an entry in a given scope with another entry,

These operations are provided in the package Parser_Symbol_Table. The next sections discuss them in more detail.

1.3 Determining the Current Scope

The routine

```
function Current_Scope return Symbol_Table;
```

returns the current table under consideration.

1.4 Creating a New Scope

To create a new scope, the programmer invokes the following routine:

```
procedure New_Scope(name);
```

where 'name' is the name of the Ada program unit with which the new scope is to be associated. This procedure creates a symbol table with the given name and assigns the current scope as the parent table of the new table. It then enters the newly created scope. For example, suppose procedure P contains a subprocedure P1, Suppose one is currently in the scope of P. In order to create the symbol table for P1, one would call New_Scope with the designator of P1 as the parameter. This would create a symbol table for P1, the would place the search cursor in the scope of this new table.

1.5 Changing the Current Scope

In addition to the New_Scope routine, there are three routines for entering a previously defined scope. These are

```
procedure Enter_Scope(k);
procedure Leave_Scope;
procedure Enter_Associated_Scope(n);
```

The procedure Enter_Scope is used for entering a previously defined subscope of the current scope. In this procedure 'k' is the name of the scope being entered. This routine simply enters the given scope.

The procedure Leave_Scope simply enters the parent scope of the current scope; thus, a call to Enter_Scope, followed by a call to Leave_Scope, will result in the current scope being the original scope.

The procedure Enter_Associated_Scope is used for entering the scope associated with a given Ada program unit. The parameter 'n' is not a symbol key but, rather, an AdaTran node that corresponds to a particular Ada program unit. The function Enter_Associated_-Scope allows the user to go directly to a given scope, without having to go up and down a tree of nested scopes. One example where this is important is searching through the 'used' units of a given scope.

For example, suppose one wishes to enter the scope of a particular compilation unit. The parameter 'n' corresponds to the unit with which the scope is associated.

1.6 Retrieving Symbol Table Entries

Logically, retrieval of symbol table entries should follow the visibility rules of Ada. This means that the retrieval process generally should first search through the local scope, then the parent scopes (in order), the through the 'with'-ed units, then through the scopes of the 'used'-units. Furthermore, there are times when a tool may wish to look just at entries in the local segment of the current unit (for example just in the 'body', the 'private' part, or the 'specification'). At other times a given tool may be interested in searching through all parts of the current scope but not in searching through any of the parent scopes.

In Ada there can be more than one declaration with a given key visible at a given point in the program. This is the case, for example, with overloaded subprograms. In order to deal with multiple visible declarations with the same name, we have introduced the notion of a 'search cursor' that keeps track of the current position in a particular search through a symbol table. Most of the retrieval routines will have both a 'First_' and a 'Next_' version. The function whose name starts with 'First_' will find the first definition corresponding to a given key, starting with the given table. The function whose name starts with 'Next_' will find the first definitions corresponding to the given key, past the cursor position of the last retrieval.

One final note. The functions for retrieving entries all use the parameter 'k,' which represents the search key. The type of 'k' has not been specified. This is because these functions are all overloaded with respect to 'k,' with 'k' being an AdaTran_Node in the one case and 'k' being a Symbol in the other case. In the case where 'k' is an AdaTan_Node, 'k' must be of type k_Symbol or k_Character. This is to allow enumeration literals, some of which can be characters, to be entered in the symbol table. Since most keys will be symbols, and since many tools will deal exclusively with keys that are symbols, it is useful to overload the retrieval functions to allow symbols themselves, rather than just symbol nodes, as keys.

1.7 The General Search Process

The general search process is to search for all visible definitions corresponding to a given key. This facility is provided by the functions

```
function Get_First_Entry(k) return AdaTran_Node;
function Get_Next_Entry(k) return AdaTran_Node;
```

The function Get_First_Entry finds the first entry with key 'k' visible in the current scope, while Get_Next_Entry finds the first visible entry with the given key past the current search cursor position. The functions Get_First_Entry and Get_Next_Entry will both search through all definitions visible from the given table, whether in the local scope, parent scopes, 'with'-ed units, or 'used'-units.

1.8 The Local Search Process

To search locally in a given scope segment, such as a specification, a body, or a private part, one uses the functions.

```
function Get_First_Local_Entry(k) return AdaTran_Node.
function Get_Next_Local_Entry(k) return AdaTran_Node.
```

These work similarly to Get_First_Entry and Get_Next_Entry, except that the search never goes beyond the current segment.

1.9 The Unit Search Process

There are times when the user wishes to limit searching to the various segments of an Ada unit. For example, one might begin a search in the body of a package and be only interested in those definitions that occur in the body, private part, and specification. This facility is provided by the functions

```
function Get_First_Unit_Entry(k) return AdaTran_Node;
function Get_Next_Unit_Entry(k) return AdaTran_Node;
```

1.10 Modifying a Symbol Table

The basic routines for modifying a symbol table are

```
procedure Add_Entry(k, value);
procedure Remove_Entry(k, value);
procedure Replace_Entry(k, old_node, new_node);
```

These three routines only affect the current scope.

The procedure Add_Entry simply adds the entry whose key is given by the parameter 'k' and whose value is given by the parameter 'value' to the current scope. The procedure Remove_Entry deletes the entry with key 'k' and value 'value' from the current scope. Finally, the procedure Replace_Entry substitutes the value given by 'new_node' for that given by 'old_node,' under the key given by 'k.'

2. The Underlying Symbol Table Mechanism

A complete discussion of the symbol table mechanism, in all its generality, is beyond the scope of this report. This section merely points out some of the novel features of the symbol table mechanism.

2.1 The Building Blocks for the Symbol Table Mechanism

The following Ada packages make up the symbol table mechanism:

```
Associations
Low_Level_Symbol_Table_Definitions
Symbol_Table_Functions
Search_Functions
```

2.2 The Low Level Packages.

The packages Associations and Low_Level_Symbol_Table_Definitions provide the basic definitions used by all the other symbol table packages. In particular, they provide the definitions for the data types 'Symbol_Table' and 'Search,' which are basic to all the other packages.

2.3 The Package 'Symbol_Table_Functions'

The package Symbol_Table_Functions provides facilities for

- creating new symbol tables,
- associating symbol tables with program units,
- associating symbol tables with their parent and descendant tables, and
- associating symbol tables with their corresponding 'with' and 'use' clauses.

2.4 The Package 'Search_Functions'

The package Search_Functions forms the heart of the symbol table mechanism. It provides the mechanism for setting up and manipulating a 'search, on the symbol table for a given Ada program. This includes facilities for

- creating new searches,
- setting the base (or starting) table for a search,
- setting the mode (LOCAL, GLOBAL, etc.) of a search,
- handling the nesting mechanism for a search,
- setting the search key for a search,
- retrieving symbol table entries associated with a given search,
- adding, removing, and replacing entries in the base table of a given search,

Note that the word 'search' in this context refers to the actual Ada data type 'search,' which is a type of data object set up to support multiple searches through the symbol table of an Ada program.

One final remark. Most tool builders will not use the packages Symbol_Table_Functions and Search_Functions directly. Instead, they will use these packages via a simplified interface, such as that provided by the package Parser_Symbol_Table.

Appendix E

SRE/ESL Internal Representation

ELEMENTARY STATEMENT LANGUAGE INTERNAL REPRESENTATION

MEMO 2 (REVISED)
January 29, 1992

DRAFT

NAVSWC CONTRACT NO. N60921-90-C-0298

Deliverable Item #0003 - A003

COMPUTER COMMAND AND CONTROL COMPANY
2300 CHESTNUT STREET STREET
PHILADELPHIA, PA 19103

TABLE OF CONTENTS

1.	INTRODUCTION	1
2.	DATA STRUCTURE OF STATEMENT AND EXPRESSION NODE IN THE PROGRAM TREE	1
2.1.	DATA STRUCTURE OF THE PROGRAM TREE STATEMENT NODES	1
2.1.1.	LANGUAGE	3
2.1.2.	STATEMENT TYPE	3
2.1.3.	AUXILIARY	3
2.1.4.	ENCODE1 AND ENCODE2	4
2.1.5.	STRUCTURE POINTERS	5
2.1.6.	LABEL_POINTER	5
2.1.7.	EXPRESSION POINTERS	5
2.2.	DATA STRUCTURE OF EXPRESSION NODES	6
2.2.1.	EXPRESSION TYPE	7
2.2.2.	POINTERS TO BROTHER EXPRESSION	7
2.2.3.	NUMBER OF DESCENDANTS	7
2.2.4.	POINTERS TO DESCENDANTS	8
2.2.5.	NUMBER OF CHARACTERS IN STRING	8
2.2.6.	STRING	8
3.	EXECUTABLE STATEMENTS	9
3.1.	CONDITIONAL BLOCK	10
3.2.	LOOP BLOCK	11
3.3.	ASSIGNMENT STATEMENT	12
3.4.	PROCEDURE CALL	12
3.5.	MESSAGE STATEMENTS	13
3.6.	INPUT/OUTPUT	13
3.7.	INPUT/OUTPUT AUXILIARY STATEMENT	13
3.8	CONTEXT STATEMENTS	13
3.9	CONTROL TRANSFER	14
4.	DECLARATION STATEMENTS	16
4.1	PROGRAM TYPE	17
4.2	STRUCTURE TYPE	17
4.3	VARIABLE TYPE	18
4.4	PROGRAM UNIT	18
4.5	STRUCTURE DECLARATION	20
4.6	VARIABLE	21
4.7	FILE	21
4.8	COMMENT DECLARATION	21
5.	EXPRESSION NODES	22
5.1	TYPES OF EXPRESSION NODES	22
5.2	FIELDS IN EACH EXPRESSION NODE	23
5.3	TREE CONSTRUCTION EXAMPLES	31
APPENDIX:	ESL STATEMENT CODE	39

Prepared Under Contract No. N60921-90-C-0298

1. INTRODUCTION

This memo describes the program tree for storing Elementary Statement Language (ESL) programs in a tree structure in memory. Block statements are nodes that have branches which fan–out to their constituent statements. Terminal statement form the leaves of the tree. Each statement is also the root of a subtree of expression nodes that contain the arguments of the statement. This memo consists of four sections. Section 2 discusses the statement and expression node structures. Section 3 describes the structure for storing executable statements. Section 4 describes the node structure for storing ESL declaration statements. Section 5 discusses the expression nodes structure. The ESL tree is used as an intermediary in translation of source real–time programming languages into Ada. A source language program is translated first to ESL. ESL has semantics similar to those of Ada. However, the ESL tree is reorganized and modified prior to translation to Ada.

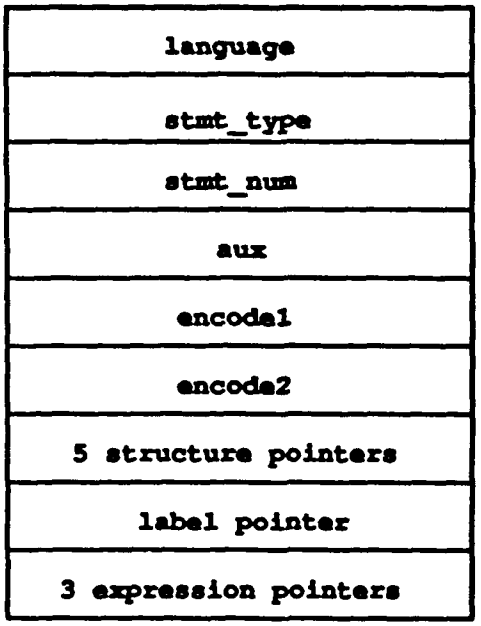
2. DATA STRUCTURE OF STATEMENT AND EXPRESSION NODE IN THE PROGRAM TREE

2.1. DATA STRUCTURE OF THE PROGRAM TREE STATEMENT NODES

This subsection describes the node structure of statements. The statement node structure is shown 'below in MODEL, C, and Ada in a structure of type node.

```
1 node is type accessed by NodePtr,
    3 language is fld(char 1),  /*ESL, EESL*/
    3 stmt_type is fld(bin fix), /*stmt type*/
    3 stmt_num is fld(bin fix),  /*stmt id number*/
    3 aux is fld(access) AuxNodePtr,  /*attribute field for future use*/
    3 encode1 is fld(char 1),  /*encode statement pointers*/
    3 encode2 is fld(char 1),  /*encode expression pointers*/
    3 father is fld(access) NodePtr,  /*immediate ancestor*/
    3 pbrother is fld(access) NodePtr,  /*previous sibling stmt*/
    3 nbrother is fld(access) NodePtr,  /*next sibling stmt*/
    3 t_son is fld(access) NodePtr,     /*then son*/
    3 e_son is fld(access) NodePtr,     /*else son*/
    3 label_pointer is field(access) EXPNodePtr,
    3 ex0 is fld(access) EXPNodePtr,
    3 ex1 is fld(access) EXPNodePtr,
    3 ex2 is fld(access) EXPNodePtr;

typedef int  stmt_kind;
typedef char languages;

struct _Node {
    languages          language;
    stmt_kind          stmt_type;
    char               stmt_num[8];          /* Statement sequence number in the program */
                                             /* It takes 8 character positions */
    struct _Auxnode *aux;         /* attribute node for future use            */
    char               encode1, encode2;
                                             /* encode1 encodes the 5 structure pointers */
                                             /* encode2 encodes the 3 expression pointers*/
    struct _Node       *father;   /* the father statement */
    struct _Node       *pbrother; /* the previous sibling statement  */
    struct _Node       *nbrother; /* the next sibling statement  */
    struct _Node       *t_son;
                                             /* the first statement of the  */
                                             /* block if the current node represents a */
                                             /* compound statement. If it is an        */
                                             /* 'if-then-else', it points to the first */
                                             /* statement of the 'then' block.         */
    struct _Node       *e_son;
                                             /* the first statement of the  */
                                             /* 'else' block if it is an 'if-then-else'*/
                                             /* statement and if there is an 'else'    */
                                             /* block, 'NULL' otherwise.               */
    struct _Expnode    *label_pointer;
    struct _Expnode    *ex0, *ex1, *ex2;
};

TYPE NODE IS RECORD
   LANGUAGE: CHARACTER;
   STMT_TYPE: INTEGER;
   STMT_NUM: INTEGER;
   AUX: AUXNODEPTR;
   ENCODE1: CHARACTER;
   ENCODE2: CHARACTER;
   FATHER: NODEPTR;
   PBROTHER: NODEPTR;
   NBROTHER: NODEPTR;
   T_SON: NODEPTR;
   E_SON: NODEPTR;
   LABEL_POINTER: EXPNODEPTR;
   EX0: EXPNODEPTR;
   EX1: EXPNODEPTR;
   EX2: EXPNODEPTR;
END RECORD;
```

This structure is graphically described as follows:



Three expression pointers ex0, ex1, and ex2 are used to store statement arguments in expression nodes. The fields of the statement structure are as follows:

2.1.1. LANGUAGE

This field is reserved for temporary and future use to denote the translation from a source language to a version of ESL. It indicates the need to reorder the programs and to use procedures that correspond to source program special functions and operating system calls. This must be done in the translation from ESL to Ada.

2.1.2. STATEMENT TYPE

Statement types are discussed in Section 3 for executable statements and in Section 4 for declaration statements. The statement type are represented in this memo by symbolic names. The statement types are listed in the tables in Section 3 and 4. The corresponding identification number of each statement type is given in Appendix I.

2.1.3. AUXILIARY

This field is reserved for temporary use in the processing of an ESL tree.

2.1.4. ENCODE1 AND ENCODE2

Encode1 and Encode2 are represented each by one character. Encode1 encodes the presence/absence of 4 structure pointers:

pbrother
nbrother
t_son
e_son

The presence/absence of each pointer above is a binary number in this order. The presence/absence of the above four pointers is encoded by one of the following 16 characters:

```
0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
```

Since every statement (except the root) has a father pointer, the presence of the father pointer is not encoded.

For instance, encode1 = '7', corresponds to the binary number

```
0111
```

from the encoding rule, we find

```
pbrother = null
nbrother /= null
   t_son /= null
   e_son /= null
```

Similarly, the character encode2 encodes use of four expression pointers:

```
label_pointer
ex0
ex1
ex2
```

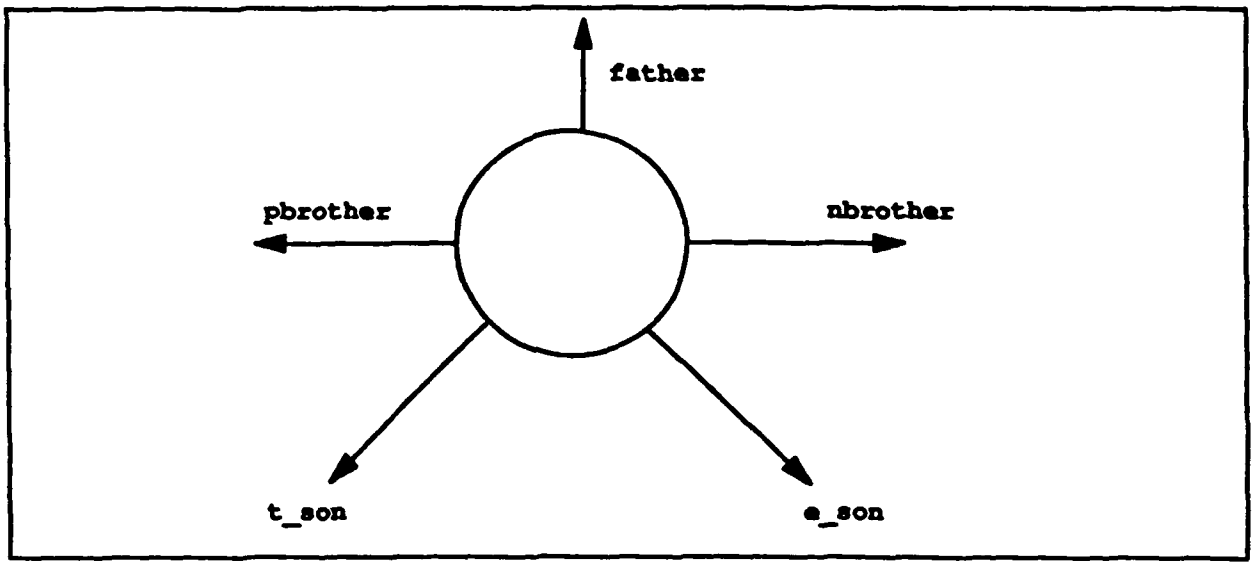
as one of the 16 characters:

```
0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
```

Encode1 and Encode2 are also used to unload the program tree to a disk file in depth first left to right order and to load back the disk file to memory and rebuild the tree.

2.1.5. STRUCTURE POINTERS

A statement is graphically portrayed as having five pointers to its neighbors, if any.



2.1.6. LABEL_POINTER

This field contains the pointer to a label expression, if any. Its presence is included in encode2.

2.1.7. EXPRESSION POINTERS

There may be as many as three main expressions representing the arguments of each statement. The existence of such expressions is coded in encode2. Each expression may consist of further subexpressions, as discussed further.

2.2. DATA STRUCTURE OF EXPRESSION NODES

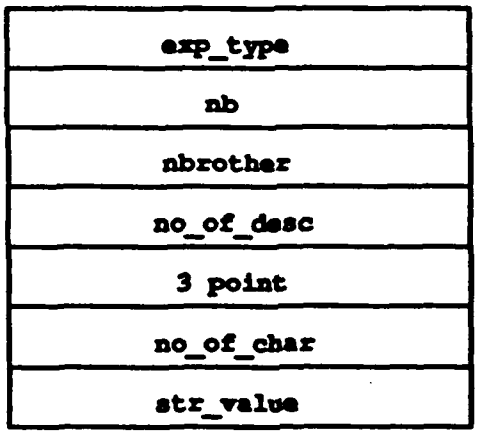
An expressions node has a structure of type expnode. Following is the definition of the structure in MODEL, C, and Ada:

```
1 expnode is type accessed by ExpnodePtr,
   3 exp_type is fld(bin fix),
   3 nb is fld(bin fix),
   3 nbrother is fld(access) ExpnodePtr,
   3 no_of_desc is fld(bin fix),
   3 point(3) is fld(access) ExpnodePtr,
   3 no_of_char is fld(bin fix),
   3 str_value is fld(char (*));  /* variable length field */

typedef int  exp_kind;              /* PIC '999'  */

struct _Expnode{
   exp_kind  exp_type;     /* Numeric code of the expression */
   int            nb;              /* It is 0 if nbrother is NULL, 1 otherwise */

   struct _Expnode *nbrother;      /* Pointer to next brother */
   int            no_of_desc;      /* Number of sons of current node */

   struct _Expnode *point[3];      /* Pointers to sons of this node */
   int            no_of_char;      /* Length of str_value     */

   char           str_value[64]; /* Variable length string value, up to 4046 */
   } ;

   TYPE EXPNODE(LEN_STR_VALUE: integer:=0)
   IS RECORD
      EXP_TYPE: INTEGER;
      NB: INTEGER;
      NBROTHER: EXPNODEPTR;
      NO_OF_DESC: INTEGER;
      POINT: EXP_VECTOR(1..3);
      NO_OF_CHAR: INTEGER;
      STR_VALUE: STRING(1..LEN_STR_VALUE);
   END RECORD;
```

where

```
TYPE EXP_VECTOR IS ARRAY(POSITIVE RANGE <>) OF EXPNODEPTR;
```

This structure is graphically described as follows:



Each field of the structure expnode is explained in the following.

2.2.1. EXPRESSION TYPE

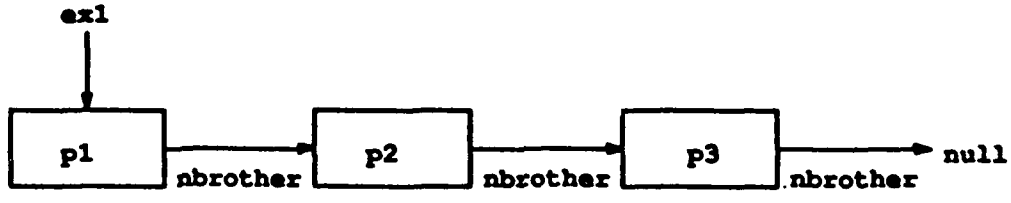
The field exp_type is an integer which identifies the type of the expression. The expression types and respective numbers are given in Section 5.

2.2.2. POINTERS TO BROTHER EXPRESSION

The field nb records the presence (nb=1) / absence (nb=1) of next expression nbrother. This enables creating a sequence of expressions. For instance, a function definition may have several formal parameters.

```
stmt_type = FCN_SPEC
ex0 -> function_name
ex1 -> parameters p1, p2, p3
ex2 -> data type of return value
```

The structure of such a statement is:



2.2.3. NUMBER OF DESCENDANTS

The field no_of_desc records the number of sons of the current node. When the node is a terminal node, it has zero descendants.

2.2.4. POINTERS TO DESCENDANTS

The field point is an array of pointers to son expression nodes. It is a three element array.

2.2.5. NUMBER OF CHARACTERS IN STRING

The string str_value has a variable length. The length (number of characters) of the str_value field is recorded in this field. A value of 0 in this field indicates that the str_value field of the expression node is not used.

2.2.6. STRING

This field str_value of the USAGE_EXPR (see Section 5 for expression types) is used to store the function of some expressions as follows.

VALUE		MEANING
COMMENT	@C	an inline comment.
DELTA	@D	precision of a fixed type.
ENUMER	@E	a list of enumerated data types.
INITIAL	@I	initial value.
LAYOUT	@Y	bit range of component.
LENGTH	@L	the length of a record in terms of bits.
NEW	@N	new instantiation of type or generic name.
PACKING	@P	word and byte information for a variable packing clause.
RANGE	@R	range of a scalar type.

3. EXECUTABLE STATEMENTS

The executable statements in ESL are listed in the following table.

	STATEMENT TYPE	STATEMENT SUBTYPE	STMT_TYPE NAME
1.	Condition	if–then–else	IF_STAT
	Block	case	CASE_STAT
		when	WHEN_STAT
2.	Loop	while	WHILE_STAT
	Block	until	UNTIL_STAT
		for	FOR_STAT
3.	Assignment Terminal	assignment	ASSIGN_STAT
4.	Procedure Call Terminal	call raise exception	CALL_STAT RAISE_STAT
5.	Message Terminal	send/receive message accept message	MSG_CALL MSG_ACCEPT
6.	Input/Output	read	READ_STAT
	Terminal	write	WRITE_STAT
7.	I/O Auxiliary	open	OPEN_FILE
	Terminal	close	CLOSE_FILE
		position	POSITION_FILE
8.	Context	with	WITH_STAT
	Terminal	use	USE_STAT
		program_separate	PACK_SEP PROC_SEP FCN_SEP TASK_SEP
		separate	SEPARATE_STAT
		pragma	PRAGMA
9.	Control Transfer	return	RETURN_STAT
	Terminal	go–to*	GOTO
		exit*	EXIT
		null*	NULL
* Statements eliminated in later processing of ESL.			

The expressions used with each statement type are discussed below.

3.1. CONDITIONAL BLOCK

A conditional block can be of types IF_STAT, CASE_STAT and WHEN_STAT.

IF_STAT statements represents:

```
IF <condition> THEN <statements1>;
               {ELSE <statements2>}
```

<condition> is a Boolean expression. The ESL statement format is:

```
stmt_type = IF_STAT;
ex0 -> <condition>;
t_son -> <statements1>;
e_son -> <statements2>, if any;
```

A CASE_STAT statement represents choice of one of several blocks <statements1>, . . ., <statementsn> according to the values value1, . . . , value n. The CASE statement contains blocks of WHEN and ELSE statements. Each of these blocks contains the respective <statementsi>:

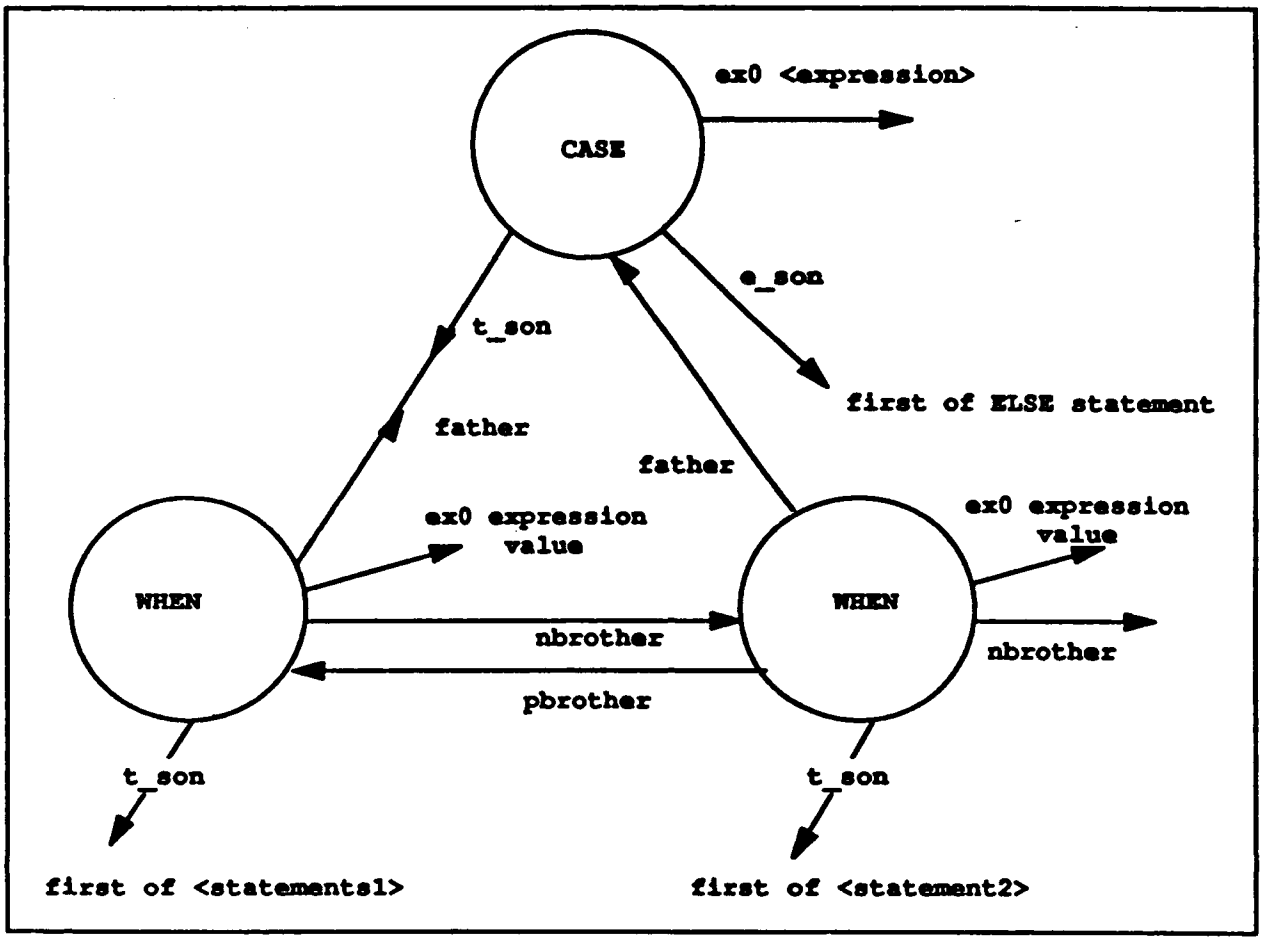
The format of the CASE statement is:

```
stmt_type = CASE_STAT;
ex0   ->  <expression>;
t_son -> <first WHEN statement>
e_son -> <first statement under the ELSE statement, if any>
```

The format of the WHEN statement is:

```
stmt_type = WHEN_STAT
ex0 -> <valuei>;
t_son -> first of statementi;
```

The CASE statement tree representation is illustrated below.



3.2. LOOP BLOCK

The loop statement has three forms: WHILE_STAT, UNTIL_STAT, and FOR_STAT.

A WHILE_STAT statement represents:

```
WHILE <condition>
```

It is followed by descendants forming the loop body.

<condition> is a boolean expression.

The ESL format is:

```
stmt_type = WHILE_STAT;
ex0 -> <condition>;
t_son -> first statement in loop body;
```

An UNTIL_STAT statement represents:

```
DO UNTIL <condition>;
```

The ESL format is:

```
stmt_type = UNTIL_STAT;
ex0 -> <condition>;
t_son -> first statement in loop body;
```

A FOR_STAT statement represents:

```
FOR <loop variable> = FROM <initial value> THRU <final value>
                                    [BY <step length>]
```

Its ESL format is:

```
stmt_type = FOR_STAT;
ex0 -> <loop variable>;
ex1 -> <initial value>, <final value>, <step length>;
t_on -> first statement in loop body;
```

3.3. ASSIGNMENT STATEMENT

An assignment always has a left hand side variable and a right hand side expression. It has the ESL format:

```
stmt_type = ASSIGN_STAT;
ex0 -> the left hand side variable(s);
ex1 -> the right hand side expression;
```

3.4. PROCEDURE CALL

This statement represents regular as well as operating system calls. The source program may call the operating system to provide certain services. Operating system calls in a source language for input/output and task communication are represented by ESL statements in the Input/Output (Section 3.5) and Message (section 3.6) categories described below respectively. Other operating system calls are handled as this type of procedure call statement.

The ESL format for a procedure call is:

```
stmt_type = CALL_STAT;
ex0 -> name of the procedure;
ex1 -> list of parameters;
```

ex1 points to a list of parameter expressions. Each parameter expression consists of a parameter name.

Operating system calls in a source language program perform a variety of functions which may not have a direct equivalent in Ada. Their call name and parameters will be stored for later analysis. Operating system calls for task messages and I/O are discussed separately below.

The ESL format for a RAISE statement is:

```
stmt_type = RAISE_STAT
ex0 -> name of exception
example:  RAISE_STAT (ERROR);
```

3.5. MESSAGE STATEMENTS

These statements are used to indicate communications between tasks. There are two statement types. MSG_CALL is used when the caller specifies the name of the other communicating task. MSG_ACCEPT is used when the communication may involve unknown other tasks. A communication must pair a MSG_CALL in one task with a MSG_ACCEPT in another task. Their ESL format is:

```
stmt_type = MSG_CALL
ex0 -> name of a procedure used to interpret a message send/receive
       operation of source program.
ex1 -> list of parameters with modes
ex2 -> a list of task and entry names

stmt_type = MSG_ACCEPT
ex0 -> name of a procedure used to interpret source program send/receive
ex1 -> list of parameters with modes
ex2 -> entry names
```

3.6. INPUT/OUTPUT

Input/Output statements represent I/O activities in the source language or its operating system. The ESL format provides for storing the operating system call name and its arguments as follows:

```
stmt_type =  READ_STAT (for input) or
                    WRITE_STAT (for output)
ex0 -> name of a procedure that interprets the operation of the source
            language and operating system I/O
ex1 -> list of parameters
ex2 -> file_name, format
```

3.7. INPUT/OUTPUT AUXILIARY STATEMENT

There are three input/output auxiliary statements: OPEN_FILE, CLOSE_FILE, and POSITION_FILE. They are stored as follows.

```
stmt_type = OPEN_FILE, CLOSE_FILE or POSITION_FILE
ex0 -> procedure name that interprets source program I/O auxiliary
       commands. Empty expression () if not applicable.
ex1 -> list of parameters
ex2 -> file name
```

3.8 CONTEXT STATEMENTS

These statements indicate that definition of a program entity is dependent on other definitions or incomplete.

WITH_STAT and USE_STAT refer to other packages. The format is

```
stmt_type = WITH_STAT or USE_STAT;
ex0 -> package names for USE_STAT
            package and program unit name for WITH_STAT ;
```

PACK_SEP, TASK_SEP, PROC_SEP and FCN_SEP are used to indicate that the body of these program units (package, task, procedure or function, respectively) is provided elsewhere and compilable separately in Ada. The format is:

```
stmt_type = PACK_STAT, TASK_SEP, PROC_SEP or FCN_SEP
```

There are no arguments. This is a terminal statement with the respective program unit specification as the parent.

The SEPARATE_STAT statement is used to indicate that the body of a program unit follows, where the specification is in another package. The format is

```
stmt_type = SEPARATE_STAT
ex0 = package name where unit specified
```

This is a terminal statement preceding the program unit body declaration.

The PRAGMA statement provides information used in the compilation. The format is:

```
stmt_type = PRAGMA
ex0 = pragma name
ex1 = list of attributes
```

3.9 CONTROL TRANSFER

A return statement returns the control from a called procedural or function to a calling procedure or function. A return statement may include an expression for a returned value.

A return statement is stored as:

```
stmt_type = RETURN_STAT;
ex0 -> expression, if any;
```

The following three statements extend ESL: GOTO, EXIT, NULL. These statements can have one or more labels. They are eliminated in later processing of ESL. Each of these statements is stored in a node statement structure, as a terminal ESL statements.

A Goto statement has its usual meaning.

```
GOTO <label>
```

The format is:

```
stmt_type = GOTO_STAT;
ex0 -> <label>
ex1 -> procedure or function name; if <label> is not in the scope of the
       immediate enclosing procedure or function.
```

An EXIT statement nested in a loop transfers control to the statement following the end of a nesting loop. If an EXIT does not have a label, the control always transfers to the end of the immediate nesting loop. If an EXIT statement has a label, the control transfers to the end of the labelled loop. The labelled loop must nest the EXIT statement.

The format is

```
stmt_type = EXIT;
ex0 -> <label>;
```

A NULL statement provides a holder for a statement label, as the destination of a GOTO statement. A NULL statement format is:

```
stmt_type = NULL;
```

4. DECLARATION STATEMENTS

The table below summarizes the ESL declaration statements.

	STATEMENT TYPE	STATEMENT SUB_TYPE	STATEMENT NAME
1.	Program Type: Block	task generic program	TASK_TYPE PACK_GEN PROC_GEN FCN_GEN
2.	Structure Type: Block	record type	RECORD_TYPE
3.	Variable Type: Terminal	variable type	VARIABLE_TYPE
4.	Program Unit: Block	system program file package task procedure function program body begin–end exception select	SYSTEM PROGRAM_FILE PACK_SPEC TASK_SPEC PROC_SPEC FCN_SPEC PACK_BODY PROC_BODY FCN_BODY TASK_BODY BEGIN EXCEPTION_DCL EXCEPTION_HNDLR SELECT
5.	Structure of Variable: Block	record	RECORD
6.	Variable: Terminal	variable constant	VARIABLE CONSTANT
7.	File: Terminal	i/o file i/o device task entry	IO_FILE IO_DEVICE TASK_ENTRY
8.	Comment:	: ordinary preprocess compiler debugging	 ORD_COMMENT PREP_COMMENT COMP_COMMENT DEBUG_COMMENT

These types of statements are further described below.

4.1 PROGRAM TYPE

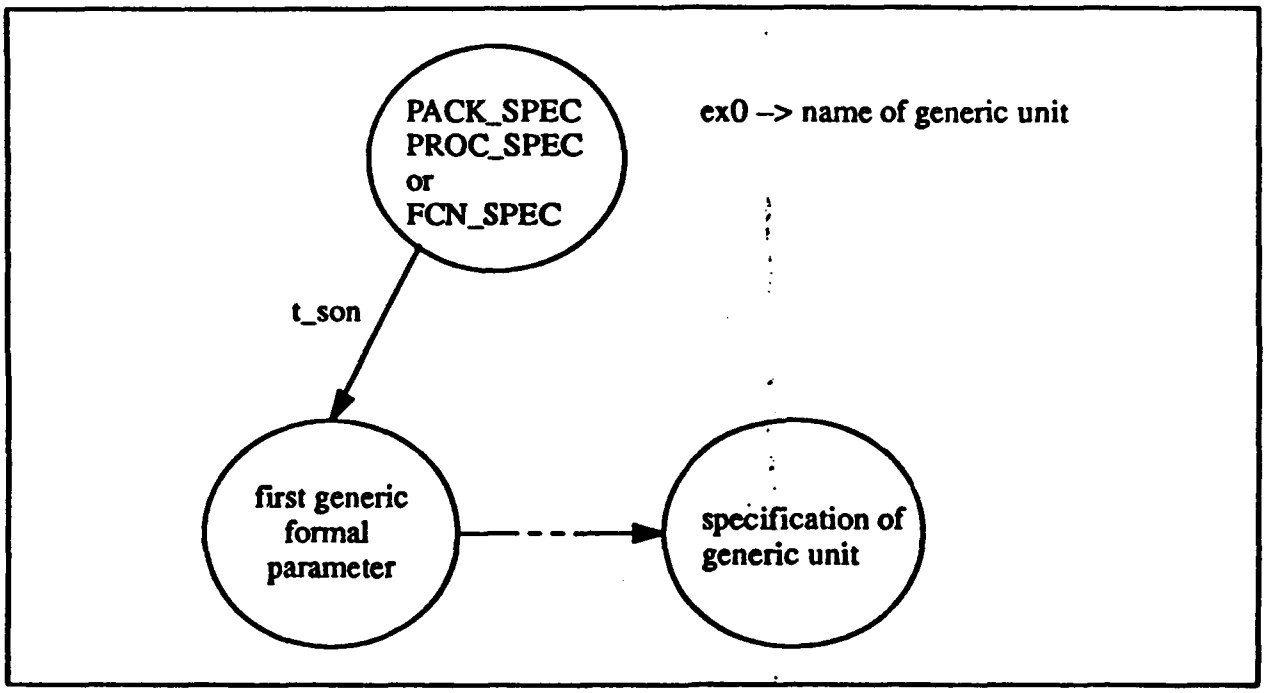
Task type is stored in the ESL program tree as follows:

```
stmt_type = TASK_TYPE
ex0 -> type name;
```

There are three generic statement types for package: PACK_GEN, for procedure: PROC_GEN and for function: FCN_GEN. They have the following points.

```
stmt_type = PACK_GEN, PROC_GEN or FCN_GEN
ex0 -> name
t_son -> first generic formal parameter
youngest sibling of t_son -> specification of generic program unit
```

This is illustrated in the figure below.



4.2 STRUCTURE TYPE

A record type declaration which has the following format:

```
stmt_type = RECORD_TYPE;
ex0 -> type name;
ex1 -> length
t_son -> first entity of the record;
```

4.3 VARIABLE TYPE

The variable type declaration is stored as:

```
stmt_type = VARIABLE_TYPE;
ex0 -> type name;
ex1 -> type definition, range, enumeration type values,
       initial value, length, packing and layout.
ex2 -> dimension ranges, if any;
```

4.4 PROGRAM UNIT

A program unit declaration is a block statement. It denotes begin–end, a system, a subsystem, a package, a task, a procedure or a function.

Begin_end has the following format:

```
stmt_type = BEGIN;
t_son -> first statement in the block;
```

A system or subsystem head is stored as:

```
stmt_type = SYSTEM;
ex0 -> system or subsystem name;
t_son -> first statement in a system;
```

A package or a task do not have parameters. Their format is

```
stmt_type = PACK_SPEC or TASK_SPEC
ex0 -> name, [name of generic package being instantiated]
```

Their body block has a similar format:

```
stmt_type = PACK_BODY or TASK_BODY
ex0-> name
t_son -> first statement
```

Note: there is no PACK_BODY of the package instantiate a generic package.

A function may have multiple IN mode parameters and returns a value. A procedure may have none or multiple IN, OUT and INOUT mode (including no value at all).

A function format is:

```
stmt_type = FCN_SPEC;
ex0 -> function name, [name of generic function being instantiated];
ex1 -> formal parameters; (or generic formal parameters
                if the function is an instance of a generic function);
ex2 -> type of return value;
```

A function body is stored similarly as:

```
stmt_type = FCN_BODY;
ex0 -> function name;
ex1 -> input formal parameters, names and types
ex2 -> type of return value;
t_son -> first statement;
```

Note: there is no function body if it is an instantiation of a generic function.

A procedure is stored as:

```
stmt_type = PROC_SPEC;
ex0 -> procedure name;
ex1 -> formal parameter name, mode, and type or generic formal paramaters
               if the function is an instance of a generic function;
```

The body of a procedure is:

```
stmt_type = PROC_BODY;
ex0-> procedure name;
ex1-> formal parameter name, types, mode and default value;
t_son -> first statement;
```

Note: there is no procedure body if it is an instance of a generic procedure.

The storage of a parameter in a function or a procedure declaration is further explained below.

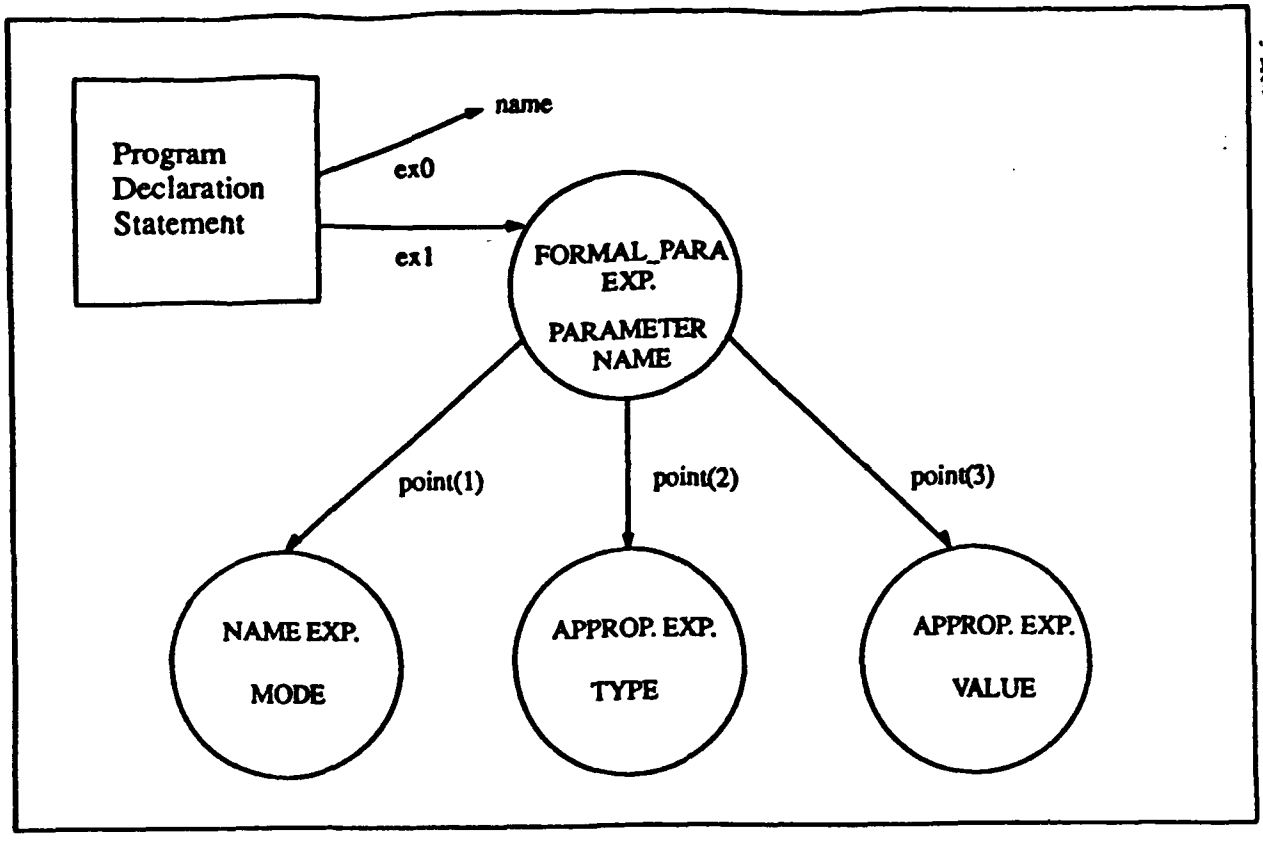
Each formal parameter may have a name, a mode, a data type, and a default value. These associated attributes are stored in expression data structure expnodes as follows:

ex1-> expnode exp_type:	FORMAL_PARA;
nbrother:	points to next parameter;
no_of_desc:	3;
point(1):	points to a NAME expnode which contains the parameter mode; That is, one of "IN", "OUT", or "INOUT";
point(2):	points to a NAME expnode, which contains the data type;
point(3):	points to an expression expnode, which is the default value of the parameter;
no_of_char:	length of the parameter name;
str_value:	parameter name;

The formats for EXCEPTION and SELECT are

```
stmt_type = EXCEPTION
example: EXCEPTION;
(descendants are the WHEN <conditions>)

stmt_type = SELECT
example: SELECT
```

This is illustrated below:



4.5 STRUCTURE DECLARATION

A record declaration is of a single or an array of records. This declaration is stored in the program tree as:

```
stmt_type = RECORD;
ex0 -> record name;
ex1 -> type, length;
ex2 -> dimension ranges;
    (if ex2=null, it represents a single record);
t_son -> first field of the record;
```

The fields are stored as descendents of the record.

4.6 VARIABLE

There are two declarations in this category: variable and constant declarations. They are stored in the program tree as follows.

variable:

```
stmt_type = VARIABLE;
ex0 -> variable name;
ex1 -> type, range, initial value, packing, length;
ex2 -> dimension ranges;
    (if ex2=null, it represents a single variable);
```

constant:

```
stmt_type = CONSTANT;
ex0 -> constant name;
ex1 -> type, value, packing, length;
ex2 -> dimension ranges;
    (if ex2=null, it represents a single constant);
```

4.7 FILE

A file declaration of a file is stored as:

```
stmt_type = IO_FILE, IO_DEVICE;
ex0 -> file name;
ex1 -> list of parameters;
ex2 -> <file type>
      <file type> could be 'sequential', 'post',
      'mail', 'isam', 'rel', 'screen', 'direct' or others used in the
            source language or operating system.
```

A task entry is declared as:

```
stmt_type = TASK_ENTRY
ex0 -> name
ex1 -> list of parameters, modes and types.
```

4.8 COMMENT DECLARATION

A comment may originate in a user comment, a keyword or comment in the source language program. Additionally, a source language keyword may be stored as a comment expression. It may affect the translation of a program from ESL to Ada.

There are four kinds of comments: ordinary user comment and source language preprocessor command, compiler command, or debugging command:

Their format is

```
stmt_type =  ORD_COMMENT, PREP_COMMENT COMP_COMMENT,
             or DEBUG_COMMENT
ex0 -> comment
```

5. EXPRESSION NODES

5.1 TYPES OF EXPRESSION NODES

The table below describes the type of expnodes.

Logical Expressions

Code	Expr Name	Operation	Operator	Example
1	OR_EXPR	inclusive disjunction	OR	a OR b
2	XOR_EXPR	exclusive disjunction	XOR	a XOR b
3	AND_EXPR	conjunction	AND	a AND b
4	NOT_EXPR	logical negation	NOT	NOT a

Relational Expressions

Code	Expr Name	Operation	Operator	Example
11	GT_EXPR	greater than	>	a > b
12	GE_EXPR	greater than or equal to	>=	a >= b
13	EQ_EXPR	equal to	=	a = b
14	NE_EXPR	not equal to	/=	a /= b
15	LT_EXPR	less than	<	a < b
16	LE_EXPR	less than or equal to	<=	a <= b

Arithmetic Expressions

Code	Expr Name	Operation	Operator	Example
21	PLUS_EXPR	addition identity	+ +	a + b +3, +a
22	MINUS_EXPR	subtraction negation	– –	a – b –22.5, – a
23	TIMES_EXPR	multiplication	*	a * b
24	DIV_EXPR	division	/	a / b
25	EXPNT_EXPR	exponentiation	**	a ** b
26	MOD_EXPR	modulus	MOD	a MOD b
27	REM_EXPR	remainder	REM	a REM b
28	ABS_EXPR	absolute value	ABS	ABS a

String Concatenation

Code	Expr Name	Operation	Operator	Example
31	CONCAT_EXPR	concatenation	&	a & b

Miscellaneous

Code	Expr Name	Operation	Operator	Example
41	PAREN_EXPR	parentheses	( )	(a+b)
42	SUBSCR_EXPR	subscripts	( )	a (i, j, k)
43	FUNCTION_EXPR	function	( )	f (a,b)
44	QUALIF_EXPR	qualification	.	file1.field1
45	ATTR_EXPR	attribute	'	m_integer' image
46	DOTS_EXPR	range	. .	1 . . 10, a . . b
47	COMMA_EXPR	delimiter, separation	,	f (a, b, c), a(i, j, k)
48	FORMAL_PARA	formal parameter clause		p1: in, integer
49	USAGE_EXPR	defines attributes		@e: red, blue

Terminal Nodes

Code	Expr Name	Operation	Example
61	STRING_CONST	charcter string	"abcdefg"
62	NUMBER_CONST	numeric constant	3.14
63	NAME	name	abc, m1

5.2 FIELDS IN EACH EXPRESSION NODE

Of the 7 fields in the expression node structure, 'nb' and 'nbrother' are not used for the purpose of storing expressions per se. They are used to indicate the existence of other related expressions. Normally, if an expression has an 'nbrother', the 'nb' field of the root node of the expression is set to 1, and the 'nbrother' field points to its 'nbrother' expression. Otherwise they are 0 and NULL respectively. Therefore, in the following description, 'nb' and 'nbrother' are not mentioned.

Logical Expressions

1. OR_EXPR (inclusive disjunction): <expr1> OR <expr2>

```
exp_type  =  1  (OR_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

2. XOR_EXPR (exclusive disjunction): <expr1> XOR <expr2>

```
exp_type  =  2  (XOR_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

3. AND_EXPR (conjunction): <expr1> AND <expr2>

```
exp_type  =  3  (AND_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

4. NOT_EXPR (logical negation): NOT <expr>

```
exp_type  =  4  (NOT_EXPR)
no_of_desc  =  1
point(1)  =  <expr>  subtree
point(2)  =  null
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

Relational Expressions

1. GT_EXPR (greater than): <expr1> > <expr2>

```
exp_type  =  11  (GT_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

2. GE_EXPR (greater than or equal to): <expr1> >= <expr2>

```
exp_type  =  12  (GE_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

3. EQ_EXPR (equal to): <expr1> = <expr2>

```
exp_type  =  13  (EQ_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

4. NE_EXPR (not equal to): <expr1> /= <expr2>

```
exp_type  =  14  (NE_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

5. LT_EXPR (less than): <expr1> < <expr2>

```
exp_type  =  15  (LT_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

6. LE_EXPR (less than or equal to): <expr1> <= <expr2>

```
exp_type  =  16  (LE_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

Arithmetic Expressions

1. PLUS_EXPR (addition, binary operation): <expr1> + <expr2>

```
exp_type   =  21   (PLUS_EXPR)
no_of_desc  =  2
point(1)   =  <expr1>  subtree
point(2)   =  <expr2>  subtree
point(3)   =  null
no_of_char  =  0
str_value   =  empty  string
```

2. PLUS_EXPR (identity, unary operation): + <expr>

```
exp_type   =  21   (PLUS_EXPR)
no_of_desc  =  1
point(1)   =  <expr>  subtree
point(2)   =  null
point(3)   =  null
no_of_char  =  0
str_value   =  empty  string
```

3. MINUS_EXPR (subtraction, binary operation): <expr1> - <expr2>

```
exp_type   =  22   (MINUS_EXPR)
no_of_desc  =  2
point(1)   =  <expr1>  subtree
point(2)   =  <expr2>  subtree
point(3)   =  null
no_of_char  =  0
str_value   =  empty  string
```

4. MINUS_EXPR (negation, unary operation): - <expr>

```
exp_type   =  22   (MINUS_EXPR)
no_of_desc  =  1
point(1)   =  <expr>  subtree
point(2)   =  null
point(3)   =  null
no_of_char  =  0
str_value   =  empty  string
```

5. TIMES_EXPR (multiplication): <expr1> * <expr2>

```
exp_type   =  23   (TIMES_EXPR)
no_of_desc  =  2
point(1)   =  <expr1>  subtree
point(2)   =  <expr2>  subtree
point(3)   =  null
no_of_char  =  0
str_value   =  empty  string
```

6. DIV_EXPR (division): <expr1> / <expr2>

```
exp_type  =  24  (DIV_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

7. EXPNT_EXPR (exponentiation): <expr1> ** <expr2>

```
exp_type  =  25  (EXPNT_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

8. MOD_EXPR (modulus): <expr1> MOD <expr2>

```
exp_type  =  26  (MOD_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

9. REM_EXPR (remainder): <expr1> REM <expr2>

```
exp_type  =  27  (REM_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

10. ABS_EXPR (absolute value): ABS <expr>

```
exp_type  =  28  (ABS_EXPR)
no_of_desc  =  1
point(1)  =  <expr>  subtree
point(2)  =  null
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

String Concatenation

1. CONCAT_EXPR (concatenation): <expr1> & <expr2>

```
exp_type  =  31  (CONCAT_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

Miscellaneous Expressions

1. PAREN_EXPR (parentheses): (<expr>)

```
exp_type  =  41  (PAREN_EXPR)
no_of_desc  =  1
point(1)  =  <expr>  subtree
point(2)  =  null
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

2. SUBSCR_EXPR (subscripted variables): <expr1>(<expr2>)

```
exp_type  =  42  (SUBSCR_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree,  the  variable
point(2)  =  <expr2>  subtree,  the  subscripts
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

3. FUNCTION_EXPR (function calls): <expr1>(<expr2>)

```
exp_type  =  43  (FUNCTION_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree,  the  function  name
point(2)  =  <expr2>  subtree,  the  actual  parameters
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

4. QUALIF_EXPR (qualification): <expr1> . <expr2>

```
exp_type  =  44  (QUALIF_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree,  such  as  record  name
point(2)  =  <expr2>  subtree,  such  as  component  in  record
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

5. ATTR_EXPR (attribute): <expr1> ' <expr2>

```
exp_type  =  45  (ATTR_EXPR)
no_of_desc  =  2
```

```
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

6. DOTS_EXPR (range): <expr1> .. <expr2>

```
exp_type  =  46  (DOTS_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree
point(2)  =  <expr2>  subtree
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

7. COMMA_EXPR (delimiter, separation): <expr1> , <expr2>

```
exp_type  =  47  (COMMA_EXPR)
no_of_desc  =  2
point(1)  =  <expr1>  subtree,  subscript  or  actual  parameter
point(2)  =  <expr2>  subtree,  subscripts  or  actual  parameters
point(3)  =  null
no_of_char  =  0
str_value  =  empty  string
```

8. FORMAL_PARA (formal parameters): [<expr1>] [<expr2>] [<expr3>] [<expr4>]

```
where  <exp1>  =  formal  parameter  name
          <exp2>  =  mode,  'IN',  'OUT',  or  'INOUT'
          <exp3>  =  parameter  type  name
          <exp4>  =  default  parameter  value,  may  or  may  not
                     be  present

exp_type  =  48  (FORMAL_PARA)
no_of_desc  =  3  if  <exp4>  present,  2  if  not
point(1)  =  <expr2>  subtree,  the  mode
point(2)  =  <expr3>  subtree,  the  type  name
point(3)  =  <expr4>  subtree,  the  default  value  if  present
                      null  if  absent
no_of_char  =  length  of  the  formal  parameter  name,  <expr1>
str_value  =  the  formal  parameter  name
```

9. USAGE_EXPR (expression usage indication): @C: <expr>
   where C is a single character indicating the usage of <expr>.

```
exp_type  =  49  (USAGE_EXPR)
no_of_desc  =  1
point(1)  =  <expr>  subtree
point(2)  =  null
point(3)  =  null
no_of_char  =  length  of  the  character  string  in  'str_value'
        str_value  "COMMENT"          if      C='C'
                   "DELTA"            if      C='D'
                   "ENUMER"           if      C='E'
                   "DIGIT"            if      C='G'
                   "INITIAL"          if      C='I'
                   "LENGTH"           if      C='L'
                   "NEW"              if      C='N'
                   "PACKING"          if      C='P'
                   "RANGE"            if      C='R'
                   "LAYOUT"           if      C='Y'
```

For each comment a usage expression node (exp_type=USAGE_EXPR) with a string constant node (exp_type=STRING_CONST) has its only descendent (point(1)), which contains the comment as its 'str_value'.

In general, the usage expression of the comment is 'pbrother' (before) or 'nbrother' (after) of the neighboring expression node which has higher precedence.

Terminal Nodes

10. STRING_CONST (character strings): "abc xyz"

```
exp_type  =  61  (STRING_CONST)
no_of_desc  =  0
point(1)  =  null
point(2)  =  null
point(3)  =  null
no_of_char  =  length  of  str_value,  not  including  quotes,
                          7  in  this  example
str_value  =  character  string  "abc  xyz"
```

11. NUMBER_CONST (numbers): 3.1416

```
exp_type  =  62  (NUMBER_CONST)
no_of_desc  =  0
point(1)  =  null
point(2)  =  null
point(3)  =  null
no_of_char  =  length  of  str_value,  6  in  this  example
str_value  =  character  string  "3.1416"
```

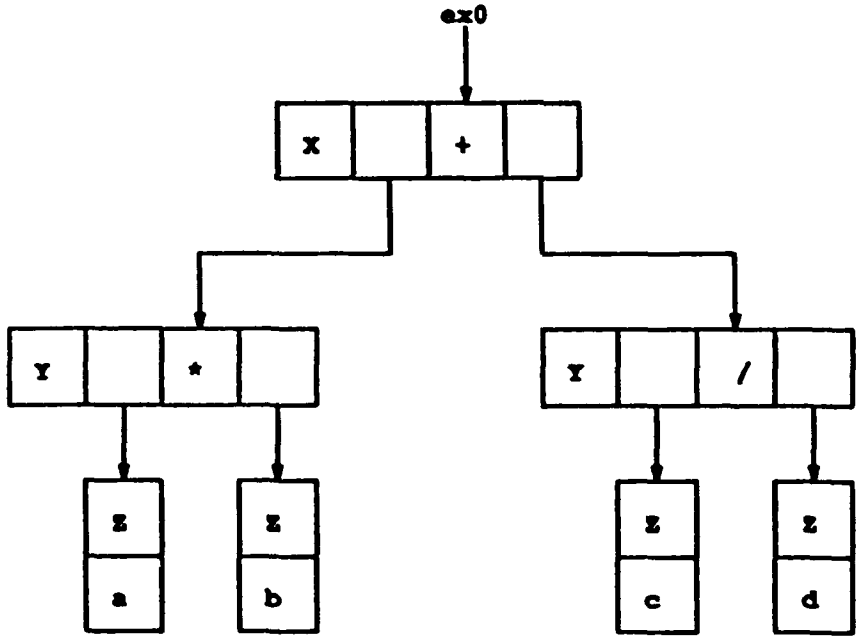
12. NAME (names): MATRIX_3

```
exp_type  =  63  (NAME)
no_of_desc  =  0
point(1)  =  null
point(2)  =  null
point(3)  =  null
no_of_char  =  length  of  str_value,  8  in  this  example
str_value  =  character  string  "MATRIX_3"
```

5.3 TREE CONSTRUCTION EXAMPLES

Example 1

In the following, an expression, a*b+c/d, is used to illustrate how an expression subtree is constructed. A horizontal rectangle represents a non-terminal node; while a vertical rectangle represents a terminal node. Each small box in the rectangle represents a field in the structure. A field from which an arrow comes out means a pointer; otherwise, it is an integer or a character string with its value indicated. For clarity only the fields involved are indicated.

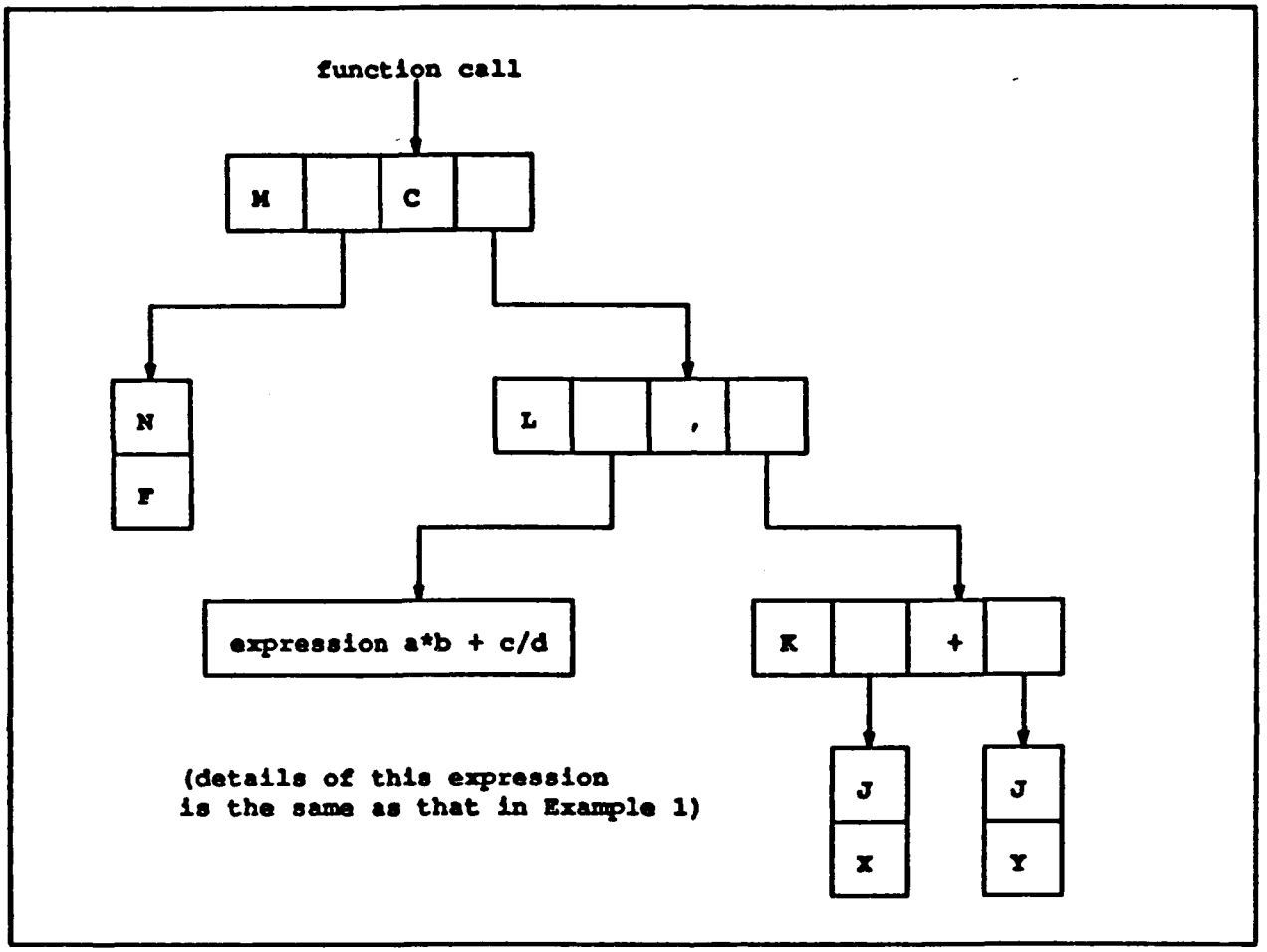


In the above diagram, x, y, and z represent the expression types as well as the operators. They have the following value:

```
x = PLUS_EXPR
y = TIMES_EXPR
z = VAR_NAME
```

Example 2

A function call, f(a*b+c/d,x+y), is used to illustrate how such an expression tree is constructed.

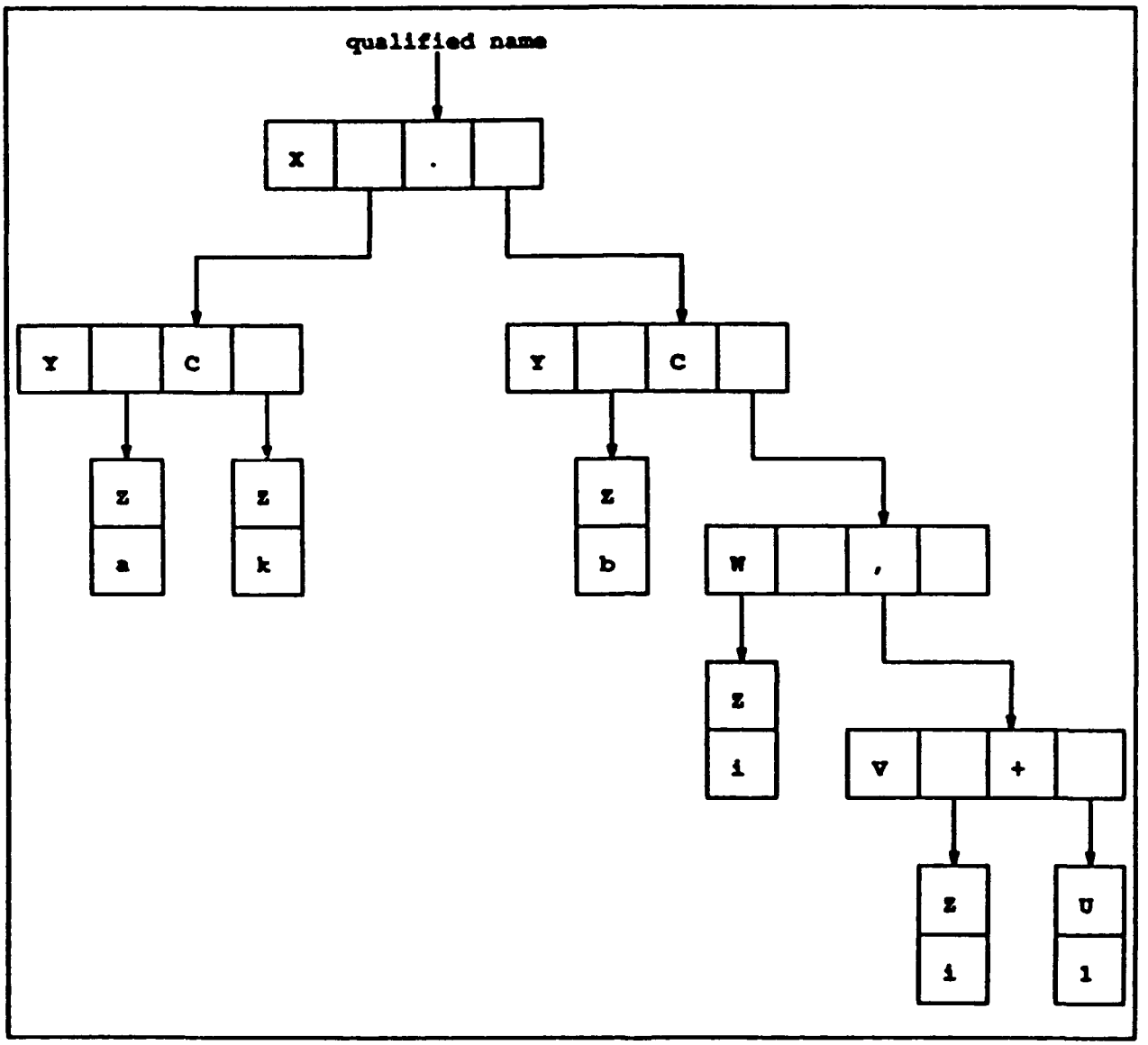


In the above diagram, M, N, L, K and J represent expresson types as follows:

M = FUNCTION_EXPR
N = PROGRAM_NAME
L = COMMA_EXPR
J = VAR_NAME

Example 3

A qualified name a(k).b(i,j+1) can be stored as follows:

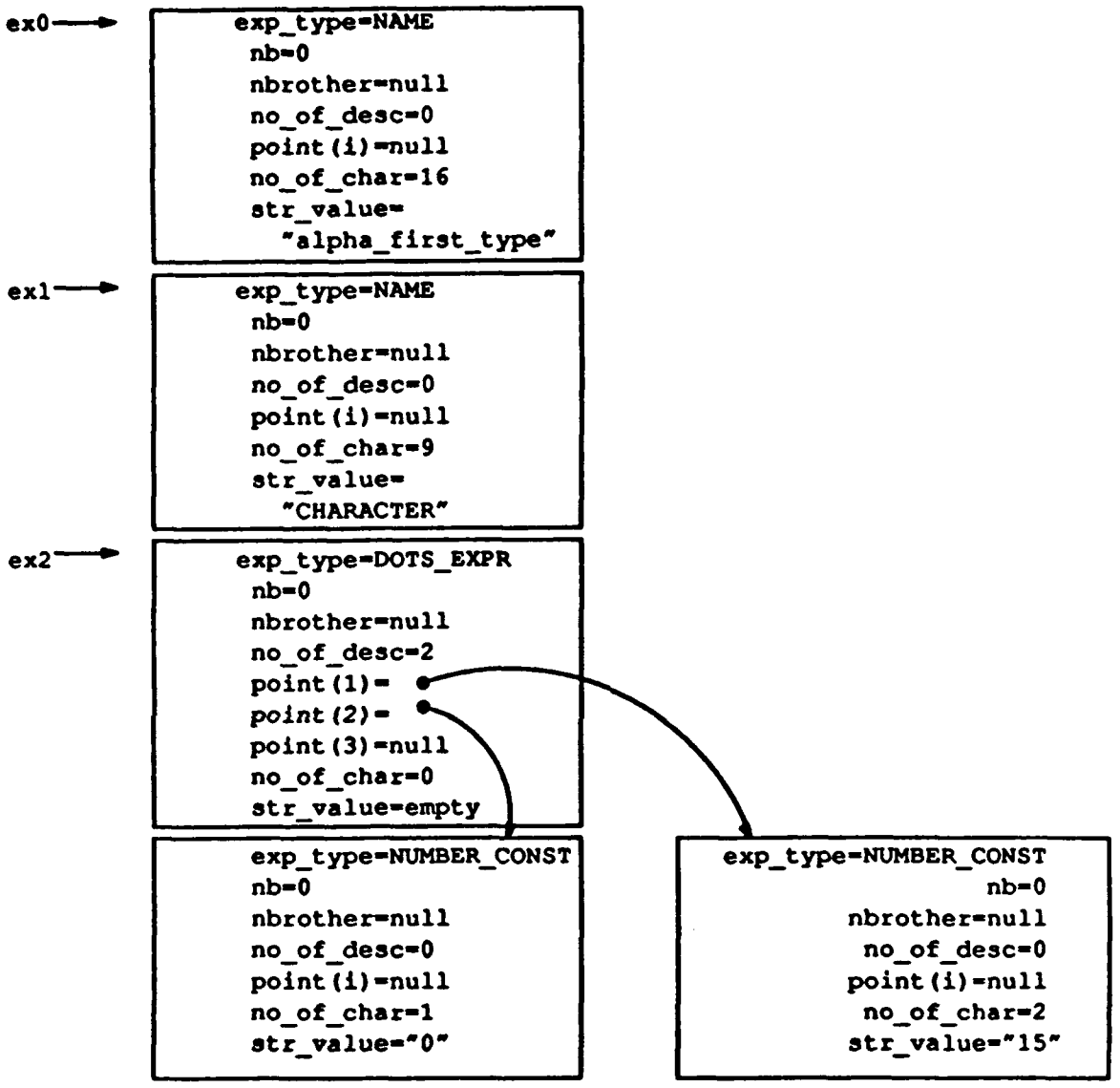


In the above diagram, X< Y< Z, W, V, and U represent expression types as follows:

```
X = QUALIF_EXPR
Y = SUBSCR_EXPR
Z = VAR_NAME
W = COMMA_EXPR
V = PLUS_EXPR
U = NUMBER_CONST
```

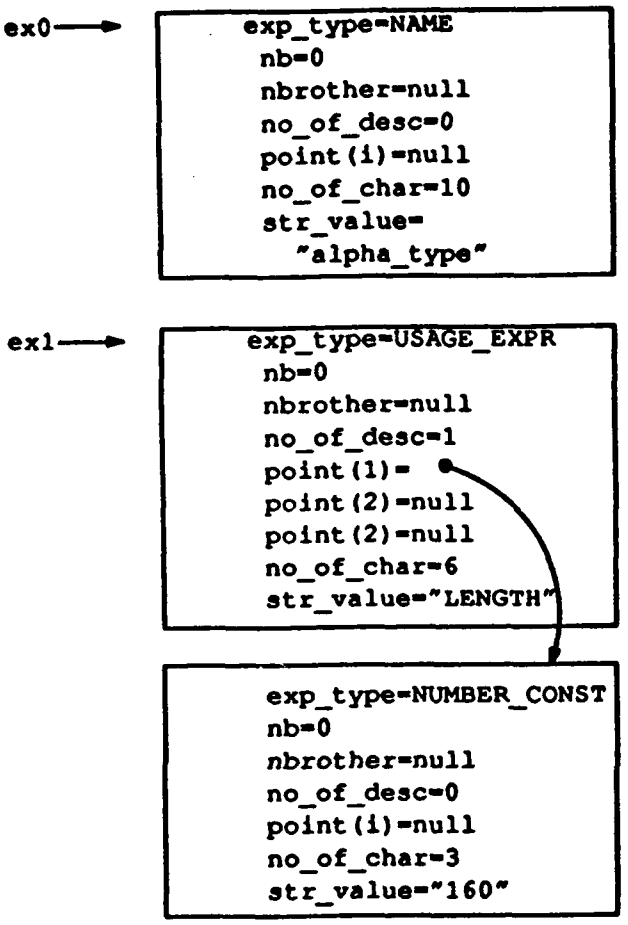
Example 4

```
VARIABLE_TYPE (alpha_first_type) IS (CHARACTER) of ((0..15));
```



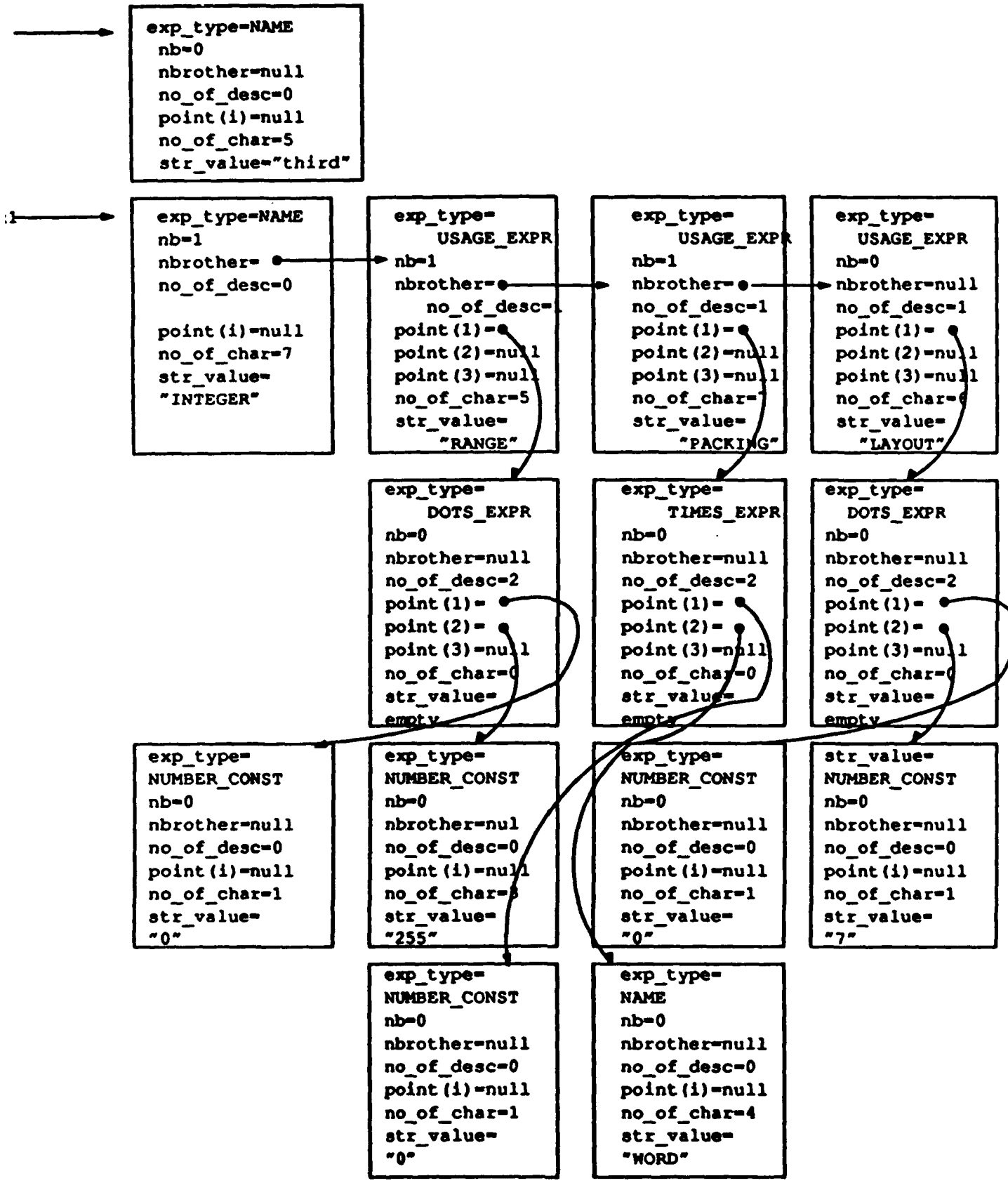
Example 5

```
RECORD_TYPE (alpha_type) IS RECORD (@L: 160);
```



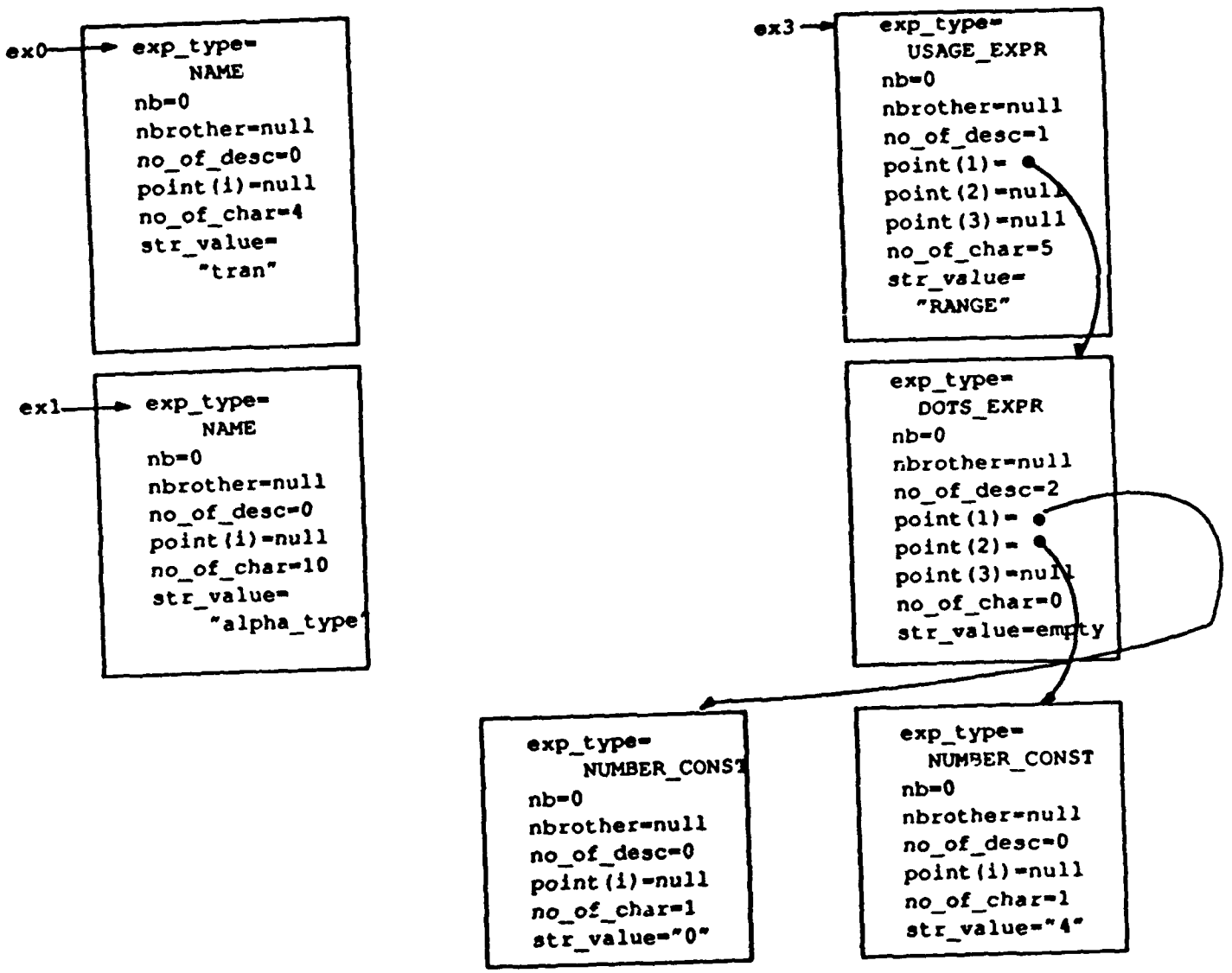
Example 6

```
VARIABLE (third) : {{INTEGER} {@R:0..255} {@P:0*WORD} {@Y:0..7}};
```



Example 7

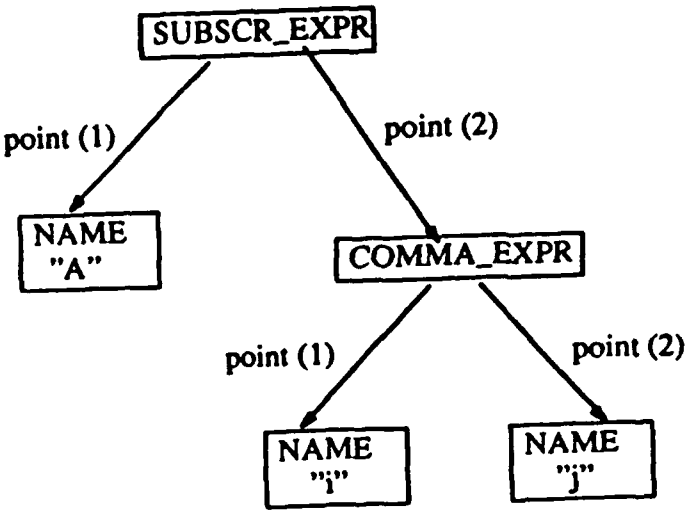
```
VARIABLE (tran) : (alpha_type) (@R:0..4);
```

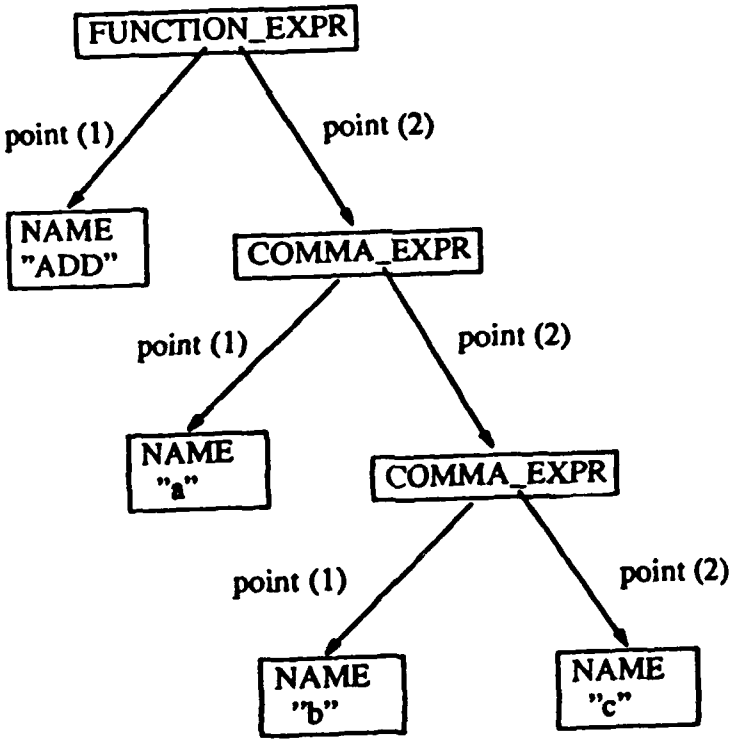


Prepared Under Contract No. N60921-90-C-0298

Example 8

The comma operator "," is used for delimiting the lists of subscripts such as those in A(i,j) or actual parameters in functions such as ADD(a,b,c). The are stored as follows





Prepared Under Contract No. N60921-90-C-0298

Appendix: ESL Statement Code

A.1. Declaration Statements

	STATEMENT TYPE	STATEMENT SUB_TYPE	STATEMENT TYPE NAME	CODE
1.	Program Type Block	task	TASK_TYPE	1
		generic	GENERIC	2
2.	Structure Type Block	record type	RECORD_TYPE	11
3.	Variable Type Terminal	variable type	VARIABLE_TYPE	21
4.	Program Unit Block	system	SYSTEM	31
		program file	PROGRAM_FILE	32
		package	PACK_SPEC	33
		task	TASK_SPEC	34
		procedure	PROC_SPEC	35
		function	FCN_SPEC	36
		program body	PACK_BODY	37
			TASK_BODY	38
			PROC_BODY	39
			FCN_BODY	40
		begin–end	BEGIN	41
		exception	EXCEPTION_DCL	42
			EXCEPTION_HNDLR	43
		select	SELECT	44
5.	Structure Block	record	RECORD	51
6.	Variable Terminal	variable	VARIABLE	61
		constant	CONSTANT	62
7.	File Terminal	i/o file	IO_FILE	71
		i/o device	IO_DEVICE	72
		task entry	TASK_ENTRY	73
8.	Comment Terminal	ordinary	ORD_COMMENT	81
		preprocess	PREP_COMMENT	82
		compiler	COMP_COMMENT	83
		debugging	DEBUG_COMMENT	84

A.2. Executable Statements

	STATEMENT TYPE	STATEMENT SUBTYPE	STMT_TYPE NAME	
1.	Condition	if–then–else	IF_STAT	101
	Block	case	CASE_STAT	102
		when	WHEN_STAT	103
2.	Loop	while	WHILE_STAT	111
	Block	until	UNTIL_STAT	112
		for	FOR_STAT	113
3.	Assignment Terminal	assignment	ASSIGN_STAT	121
4.	Procedure Call	call	CALL_STAT	131
	Terminal	raise exception	RAISE_STAT	132
5.	Message Terminal	send/receive message accept message	MSG_CALL MSG_ACCEPT	141 142
6.	Input/Output	read	READ_STAT	151
	Terminal	write	WRITE_STAT	152
7.	I/O Auxiliary	open	OPEN_FILE	161
	Terminal	close	CLOSE_FILE	162
		position	POSITION_FILE	163
8.	Context	with	WITH_STAT	171
	Terminal	use	USE_STAT	172
		program_separate	PACK_SEP TASK_SEP PROC_SEP FCN_SEP	173 174 175 176
		separate	SEPARATE_STAT	177
9.	Control Transfer Terminal	return go–to * exit* null*	RETURN_STAT GO TO EXIT NULL	181 182 183 184

* Extension eliminated in later translation.